DESIGNING AND IMPLEMENTING AN ADA LANGUAGE BINDING SPECIFICATION FOR ODMG-93

THESIS

Stephen Robert Lindsay
Captain, USAF

AFIT/GCS/ENG/94D-16

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-16

DESIGNING AND IMPLEMENTING AN ADA LANGUAGE BINDING SPECIFICATION FOR ODMG-93

THESIS
Stephen Robert Lindsay
Captain, USAF

AFIT/GCS/ENG/94D-16

Approved for public release; distribution unlimited

AFIT/GCS/ENG/94D-16

DESIGNING AND IMPLEMENTING AN ADA LANGUAGE BINDING SPECIFICATION FOR ODMG-93

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Stephen Robert Lindsay, B.S.
Captain, USAF

December, 1994

Table of Contents

Page

List of Figures

List of Tables

Acknowledgements

I would like to thank my thesis advisor, Lt Col Mark Roth, for his guidance and encouragement over the past eighteen months, and especially for making my research a top priority while simultaneously administering his full-time responsibilities at Offutt AFB, Nebraska. I'd also like to thank Jordan Kayloe for his expertise with Ada, Bob Caley for knowing everything there is to know about Unix, and both of them for taking the time to explain the details to me. My patient and omniscient system administrators, Dan Zambon and Dave Doak, made this project possible, and I could never show them the appreciation they deserve. Most importantly, I'd like to thank my wife Marissa, without whom none of this would have meant as much.

Stephen Robert Lindsay

AFIT/GCS/ENG/94D-16

Abstract

Object-oriented database management systems (ODBMSs) represent the latest advancement in database technology, combining the reusability and maintainability of the object-oriented programming paradigm with the ability to efficiently store and retrieve a wide range of data types as well as code to manipulate stored data. Unfortunately, programmers developing software in the Ada programming language do not have the ability to interface to object databases without significant customized code development. One important reason for this has been the absence of a standard defining the constructs accessing ODBMS functionality.

This thesis documents the creation of an Ada language binding to the ODMG-93 standard for object database interfaces. Key portions of the binding are then implemented for the Itasca and ObjectStore ODBMSs written in Lisp and C, respectively. This work achieved the goals of a complete, portable, and transparent object database interface for Ada. To satisfy transparency a preprocessor was assumed to exist for both implementations, and its degree of involvement was directly proportional to the degree of strong typing of the ODBMS implementation language.

DESIGNING AND IMPLEMENTING AN ADA LANGUAGE BINDING SPECIFICATION FOR ODMG-93

I. Introduction

1.1 Overview

Object-oriented database management systems (ODBMSs) represent the latest advancement in database technology, combining the reusability and maintainability of the object-oriented programming paradigm with the ability to efficiently store and retrieve a wide range of data types as well as code to manipulate stored data. Unfortunately, programmers developing software in Ada, the programming language mandated by the Department of Defense (DoD), do not have the ability to interface Ada to object databases for two reasons. First, until recently an object database standard did not exist, resulting in numerous vendors developing a wide variety of object database applications, each of which is claimed to be an ODBMS. Second, a complete and portable Ada interface package specification corresponding to the operations available in any ODBMS has yet to be produced. Until these conditions are satisfied, Ada programmers cannot easily access this powerful new technology. To date there has been limited development of "one-time" interface packages written to provide minimal ODBMS functional access satisfying only the requirements of a given software project. More than ever, Ada programmers need a complete and portable interface package providing access to any of the existing ODBMSs. This thesis documents the implementation of such an interface based upon previous research at the Air Force Institute of Technology (AFIT) and a newly proposed object database standard.

1.2 Background

As an increasing number of DoD missions require software support, the need for reusable and easily maintainable software becomes paramount. Without such software, large amounts of resources are inevitably spent duplicating previously developed capabilities that cannot be reused, typically for one of two reasons: the two capabilities require different programming language implementations, or the previous capability is implemented in the same language but in a form not easily reusable.

In 1990, Deputy Assistant Secretary of the Air Force Lloyd K. Mosemann helped solve the first problem by mandating the Ada programming language as the sole language to be used in defense software projects (12:2). With a single mandated language, a DoD organization needing a software capability can more easily reuse an existing implementation from another organization. The latter and more widespread problem of nonreusable form, however, remained unaddressed.

In the mid-1980s, the object-oriented programming paradigm first appeared as an alternative to the standard procedural and functional programming paradigms of the previous two decades. Object-oriented programming provides a natural approach to problem solving by viewing the components of a system as objects communicating with each other, rather than as tasks or functions executing in sequence (19:6–7,9). The object-oriented techniques of inheritance and aggregation help programmers develop software that is more understandable, reusable, and maintainable than its predecessors. The Ada 9X compiler, adding object-oriented constructs to the Ada language, gives Ada programmers the benefits of this programming paradigm.

Object-oriented techniques have also been applied to databases, with ODBMSs appearing as the most recent enhancement to database management systems (DBMSs). Like their programming language counterparts, these databases have proven more reusable and maintainable than their predecessors, due primarily to the ease with which database structuring changes can be incorporated and the closer match between data structures in the database and in the programming language (16:3–5).

While the Ada 9X compiler facilitates the development of object-oriented Ada software, Ada programmers still cannot write applications benefiting from the capabilities of object data management. Ada applications can use only the relational, network, or flat-file models—and not the object model—to store object-oriented data. A programmer wishing to reuse such an application and its database with only minor modifications will encounter difficulties due to the rigid structuring and limited data types of these earlier database management systems.

1.3 Approach

This thesis documents the implementation of an Ada language binding based on the ODMG-93 standard proposed by the Object Data Management Group (ODMG), and two sets of corresponding package bodies: one for the Itasca ODBMS and one for the ObjectStore ODBMS. Figure 1.1 shows the interaction of each component and the user. The Ada binding provides access from the Ada application to one of the ODBMSs, which in turn provides access to the database files.

As recommended in the design proposed by Moyers, three goals are sought for the Ada binding: completeness, transparency, and portability (14:3-5–3-6). Completeness means the Ada application can access all of the object database functionality specified in ODMG-93. Transparency means the application programmer does not need to know anything about the ODBMS implementation

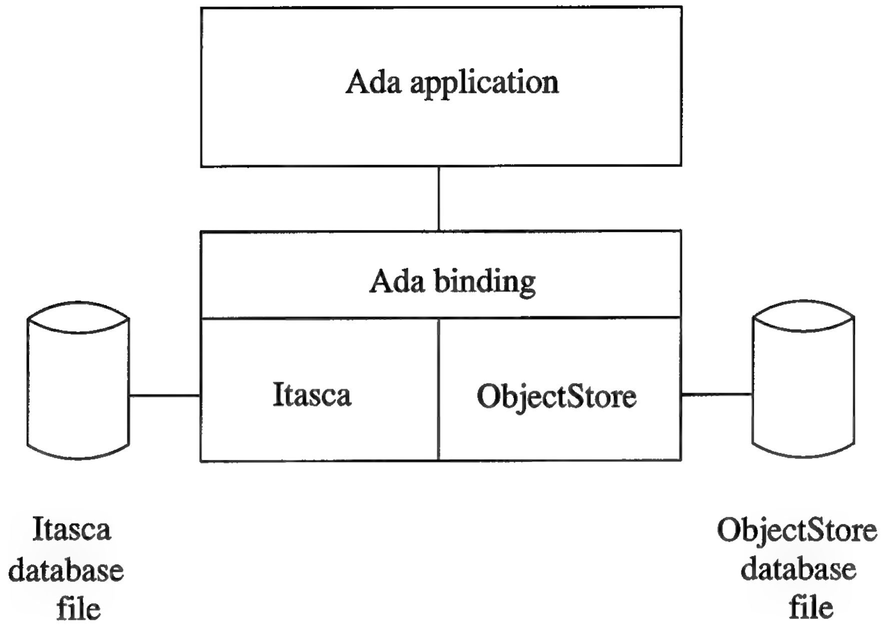

Figure 1.1 Interaction of Interface Components

language, and portability means the ODBMS package body implementations can be exchanged without modifying the application.

In addition, two goals are sought for the resulting ODBMS implementations: expandability and maintainability. If an ODBMS vendor develops its own Ada application program interface (API), this interface should be easily incorporated into the corresponding binding implementation to use the improvements of the vendor. Moreover, if the vendor changes any of its ODBMS implementation routines, the implementation must also be able to incorporate these changes as well.

An important assumption in this effort is the existence of a preprocessor to create new databases and to translate query expressions from an Ada format to the ODBMS language format. Ideally, the Ada programmer should not need to know the syntax of the ODBMS implementation language; unfortunately, every ODBMS requires the creation of databases in an interface language supported by its system. Therefore, database creation details can be made transparent to the programmer only if a preprocessor exists. By the same measure, expressions to query the database also require a preprocessor to be fully transparent. Because writing and implementing a preprocessor is a nontrivial task, and because it accounts for a only small portion of an Ada language binding implementation, a preprocessor is assumed to exist for each ODBMS implemented in this thesis. Its existence is simulated by producing the preprocessor output manually.

Additionally, the Ada binding makes use of previous research involving interfacing Ada to object databases. One example is the Ada/ObjectStore prototype interface developed by Dave Rosenberg of Object Design, Inc. (18), and extended by Li Chou at AFIT (4). This prototype interface is used as a building block in implementing the Ada binding for ObjectStore. As a result, the following assumptions apply:

1. The Ada/ObjectStore prototype interface developed by Dave Rosenberg of Object Design, Inc., is correct and needs no additional modifications.
2. Li Chou's extensions to the Ada/ObjectStore interface are correct and need no additional modifications.
3. Applications using the Ada binding will be compiled with a validated Ada compiler providing access to modules written in the C programming language.

1.4 Justification

The Ada language binding to ODMG-93 gives Ada programmers several benefits. First, because each ODBMS has its own unique strengths, an object database can be chosen for a project to exploit the strengths of a particular system. Second, if an error is detected in one ODBMS, it can be exchanged for a different system without modifying the original code, thus providing an immediate recovery capability. Most importantly, however, the binding helps Ada programmers design portable object database applications by showing them the operations and constructs they can expect to be supported by the ODBMS.

Several additional benefits are realized from this research effort. By reviewing the decisions and alternatives applicable to designing the Ada binding, a software developer may find it easier to create a new language binding to ODMG-93 than by working without an example. In addition, because the package specifications correspond to the proposed object database standard and are independent of any underlying ODBMS, they remain valid templates against which additional vendor-specific package bodies may be implemented. Finally, the package body implementations and the corresponding implementation decisions and lessons learned will facilitate these future package body implementations.

1.5 Methodology

The task of designing and implementing an Ada language binding to ODMG-93 is divided into several intermediate subtasks.

1.5.1 Specific Steps. First, the Ada binding is created by determining how each ODMG-93 construct maps into the Ada programming language. Next, the optimal method for implementing the Ada binding, without regard to a specific ODBMS, is determined. In this context, "optimal" refers to the degrees to which the expandability and maintainability goals are preserved in the resulting implementation. After determining this optimal method, ODMG-93 Object Definition Language (ODL) is implemented for Itasca and ObjectStore, followed by Object Manipulation

Language (OML) and Object Query Language (OQL) implementations, using the C programming language application program interface to access each ODBMS's functionality.

1.5.2 Materials and Equipment. The equipment used in this effort is a Sun Microsystems SPARC 2 computer, the Sun Ada compiler, and the ObjectStore and Itasca ODBMSs. No additional equipment or software is required for completing this research.

1.6 Summary

This document is comprised of six additional chapters. Chapter II, *Literature Review*, discusses background information and reviews previous work directed toward an object database interface for Ada. Chapter III, *Binding ODMG-93 to Ada*, maps each of the ODMG-93 constructs into their Ada counterparts to produce the Ada binding, and Chapter IV, *Approach Selection and ODL Implementation*, selects the optimal approach and implements Ada Object Definition Language for the Itasca and ObjectStore ODBMSs. Chapter V, *OML Implementation*, discusses the implementation of Ada Object Manipulation Language, and Chapter VI, *OQL Implementation*, discusses the implementation of Ada Object Query Language for both of these products. Finally, Chapter VII, *Conclusions and Recommendations*, presents a summary of this document and recommends several possible directions for future research.

II. Literature Review

2.1 Overview

Object database technology is the result of years of evolution in the field of database management systems. Of the different database models, object databases provide the user with the most functionality. Additionally, numerous object databases have application programming interfaces for a variety of languages, but each of these interfaces is specific to the system it accesses. This chapter reviews the origin of object databases and examines previous efforts interfacing Ada to database management systems.

2.2 Database Management System Evolution

Database management systems (DBMSs) have evolved significantly over the past thirty years. Before database systems existed, flat files were the sole method used to store and retrieve data for applications. This simple method uses the operating system file structure to create files representing the database. When the user wants to retrieve a particular record, he or she must look through each record in the corresponding file until the desired record is found, or implement an index or hash function to improve efficiency. This is the case for any storage or retrieval operation; the application is responsible for all of the database's functionality, causing the user to spend a large portion of time and effort dealing with file input and output (I/O) and data format conversion.

2.2.1 First Generation. Network and hierarchical models are considered the first generation DBMSs, where some of the database functionality is handled by the DBMS rather than by the application. First appearing in the early 1960s, these models allow the user to create collections of data by using links or pointers to records (10:555,595). While network and hierarchical models provide more efficient ways to store and retrieve records, the user is still responsible for navigating through these pointers to add, delete, and retrieve records satisfying a desired set of conditions.

2.2.2 Second Generation. The relational model was proposed in the late 1960s and marks the first DBMS model to remove the user from a substantial portion of the database's implementation details (10:95). This second generation model represents the database through the use of tables, or relations, that store the data as records in a logical format for the application to manipulate (10:53–55). In addition, the relational model is based on a formal mathematical foundation helping to ensure correctness of results and data integrity. For these reasons relational database management systems (RDBMSs) are the primary database systems used in commercial applications today.

2.2.3 Third Generation. The object model is the most recent model to be developed. This model handles even more functionality for the user, as object databases have the ability to store not only data but also code, or methods, to perform operations on the data (17:44). In addition, the object model is characterized by a greater degree of reusability and maintainability than the relational model. These benefits are consistent with nondatabase applications developed with the object-oriented programming paradigm (19:9–10).

An ODBMS stores its data as objects and relationships to other objects, rather than as relations. This accounts for a significant time and space savings over RDBMSs, because entire tables do not have to be scanned for desired record values, and because data values do not have to be stored redundantly in different tables. Instead, the desired object is accessed based on its attributes, and then any relationships in which that object participates leads the user directly to his or her result.

The object model is considered more reusable and maintainable because new attributes can be introduced more easily into an object database than into a relational database. Based on its particular attributes, a new object can automatically inherit attributes, methods, and relationships defined by existing objects. Constructs that are unique to the new object can then be defined with no redundancy. If a new attribute is added to a relation in an RDBMS, the entire relation will typically require revision, along with revisions to every record participating in that relation.

Figure 2.1 shows the increased functionality of DBMSs throughout their evolution. Object databases currently have the most functionality and hence the most value to programmers, especially to those programming in an object-oriented language. For this reason, an interface allowing access from a particular language to an object-oriented database provides the language with a powerful tool for data processing. This is precisely why efforts are currently aimed toward developing an object database interface for Ada.

2.3 Ada/ODBMS Design Efforts

A great deal of background work has facilitated the implementation of an Ada/ODBMS interface. These efforts have continued to expand, resulting in a thorough design proposal.

2.3.1 Ada/Objectstore Prototype Interface. The first individual to examine the feasibility of such an interface was Dave Rosenberg of Object Design, Inc., producers of the ObjectStore ODBMS. To fulfill previous contract requirements, Rosenberg designed a prototype interface providing a minimum set of functional access to ObjectStore for programs written in Sun Ada (18:1–9). Rosenberg's sole intent was to determine if Ada programs could indeed access ObjectStore; he did not attempt to provide an interface complete in its functionality. The design he chose was based on

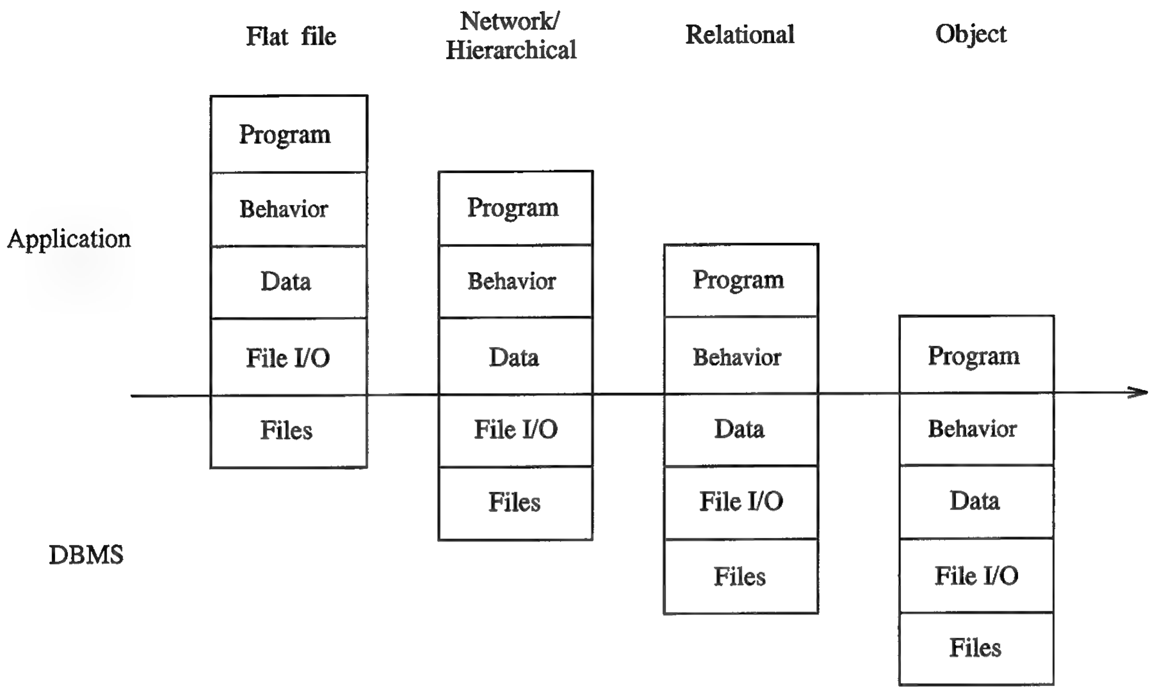

Figure 2.1 Evolution of Database Application Environments (13:52)

two facts: first, ObjectStore can be accessed by programs written in the C programming language, and second, Sun Ada has utilities allowing it to link with object code produced by the C compiler (20:109–129). Rosenberg then defined a mapping of Ada commands and data types to a subset of the corresponding commands and data types in C. Finally, he wrote several simple driver routines to successfully demonstrate how Ada could access his prototype interface. Because this was only a prototype, Ada programmers still did not have the capabilities they would need in order to take advantage of object data management.

2.3.2 Extensions to Ada/ObjectStore. Li Chou furthered the development by providing additional functionality to the prototype interface package and then conducted tests to determine if any serious performance degradation was experienced. Chou added the capability to access ObjectStore's collection facility, allowing the user to instantiate collections (such as sets, lists, bags, and arrays) of related objects for performing optimized query operations over those objects. He then wrote a driver routine creating a database and populated it with 10,000 objects to determine its functional correctness. Next, he wrote two versions of additional test routines: one version in Ada, using the interface, and one version in C, accessing ObjectStore directly. These additional routines queried the object population in various ways, carefully measuring the amount of time required, both in processor clock cycles and in user waiting time. He verified the performance of the Ada interface as typically less than ten percent slower than the direct access provided by ObjectStore's C interface (4:4-1–4-5). At this point, it became evident that an efficient Ada/ODBMS interface taking full advantage of object database technology was indeed possible.

2.3.3 Design of a Portable Ada Interface. To improve upon these efforts, Anthony Moyers added one level of abstraction and defined the necessary components for an interface not constrained to a specific ODBMS. Moyers' design consisted of a single Ada package specification and numerous package bodies, one for each particular database system (14:3-5–3-7). The different package bodies could then be interchanged, the application recompiled, and the resulting program re-executed with a new underlying ODBMS. If possible, such an implementation would prove quite powerful for two reasons. First, because each ODBMS has its own unique strengths, an object database could be chosen for a project to exploit the strengths of a particular system. Second, and more importantly, if an error were detected in one ODBMS, it could be exchanged for a different system without modifying the original code, thus providing an immediate recovery capability.

Moyers defined a set of routines for the Ada interface, and although this represented a good starting point, it had two drawbacks. First, the routines were included based on a sampling of the features common to various ODBMSs, and not on an industry standard agreed upon by the object database community. This was no fault of Moyers, as an object database standard did not exist at the time. Second, the functionality was also determined by researching what is needed in a typical software project requiring database support. Again, even though his resulting interface design seemed applicable to existing projects, it could not be verified until a standard could be produced.

2.3.4 An Object Database Standard. Fortunately, representatives of thirteen leading ODBMS vendors and other related software companies formed the Object Database Management Group (ODMG) with the primary purpose of forming a standard for object databases. The need for a standard was becoming critical; even though ODBMS technology was and still is relatively young, the different vendors were moving rapidly in their own diverse directions. As a result, ODBMS acceptance has suffered:

> To date, the lack of a standard for object databases has been a major limitation to their more widespread use. The success of relational database systems did not result simply from a higher level of data independence and a simpler data model than previous systems. Much of their success came from the standardization that they offer. The acceptance of the SQL [relational] standard allows a high degree of portability and interoperability between systems, simplifies learning new relational DBMSs, and represents a wide endorsement of the relational approach. (1:1)

Although the proposal is still in draft form, initial indications from the object database community have shown it will soon be accepted with few revisions. Some deficiencies have been identified, but most of these are due to the fact that it is not backward compatibile with the SQL relational standard (8:4).

The ODMG-93 standard defines a minimum set of constructs that an ODBMS must support in order for it to be considered a true object-oriented database management system. Furthermore, the standard describes all routines and data structures in enough detail to facilitate implementation of the Ada binding. At the same time, the level of detail is high enough for the binding developer to have the necessary flexibility. Therefore, each requirement of ODMG-93 can be incorporated into the Ada binding package specifications and then implemented in the corresponding package bodies. Consistent with Moyers' design, features exceeding this minimal standard can be implemented as annexes to the original package specification, thus taking full advantage of each vendor's unique ODBMS enhancements and preventing the interface package from becoming nothing more than a "least common denominator" of functionality.

2.4 Summary

The past thirty years have seen significant improvements in database technology. Object databases have the most functionality of all DBMSs and therefore the most value to programmers. The original Ada/ObjectStore prototype interface shows it is possible to provide Ada programmers with access to an object database. Li Chou's extensions prove such an interface can be made functional without sacrificing performance. Moyers' interface package specification helps identify the routines to be made available to Ada programmers, and the ODMG-93 standard provides a detailed set of constructs that must be supported by any true ODBMS. The next chapter documents the development of an Ada binding to ODMG-93.

III. Binding ODMG-93 to Ada

3.1 Overview

This chapter discusses the issues pertinent to producing an Ada language binding to the ODMG-93 construct specifications. Two possible designs are identified and the optimal design is selected. The ODMG-93 constructs are then mapped to their Ada counterparts to produce the Ada binding.

3.2 ODMG-93 Integration with Ada

The object database standard integrates well with Ada's packaging approach to encapsulation and information hiding; a single package specification independent of any vendor's ODBMS can be used to provide access to a category of ODMG-93 constructs using the syntax set forth in the standard. The corresponding vendor-specific package bodies can implement them for a particular system. Ada's **use** clauses can then be incorporated to make the constructs appear as a natural extension to the language.

3.3 Determining the Language Binding Approach

The most important decision for the Ada ODMG-93 language binding is the overall approach to object definition and manipulation, as this decision then dictates the definition of all remaining ODL and OML constructs. Two approaches are available: *address-based* and *generic-based.*

3.3.1 Address-based. The address-based approach uses pointers by defining type `Object` as a subtype of the platform-dependent `System.Address`. Figure 3.1 shows a typical definition of an ODMG-93 construct using this approach.

```
subtype Object is System.Address;

procedure Name_Object(An_Object : in out Object;
                      Name      : in     String);
```

Figure 3.1 Example Address-based ODMG-93 Operation

```
generic
  type User_Object_Type is private;
package Persistent_Naming is

  procedure Name_Object(An_Object : in out User_Object_Type;
                        Name      : in     String);

  function Lookup_Object(Name : in String) return User_Object_Type;

end Persistent_Naming;
```

Figure 3.2 Example Generic-based ODMG-93 Operation

This technique allows the definition of address parameters to and from the various ODMG-93 constructs without regard to the data they reference. If, however, an address referencing an invalid or inappropriate data structure is passed to a database operation, an exception is raised. This approach removes the need to overload operations based on all the valid data types the user may specify; however, it does require the majority of object related error-checking to be performed at runtime rather than at compile time.

3.3.2 Generic-based. An alternative to the address-based method is the generic-based method. With this technique, all database operations are implemented as generics and are accessed by their instantiations. For example, a package providing access to persistent name manipulation may be defined as shown in Figure 3.2.

```
package Person_Naming is new Persistent_Naming(Person.Object);
use Person_Naming;

A_Person : Person.Object := Person.Create
  (A_Database, Last_Name => "Jones");

Name_Object(A_Person, "JJ");
```

Figure 3.3 Example Generic-based Instantiation

An instantiation of the generic would then allow the naming constructs to be visible for the type specified by the user. Figure 3.3 shows a typical instantiation.

To alleviate the user from instantiating what could easily be a large number of generics, the ODL preprocessor could automatically instantiate all Ada binding operations for each persistent-capable class by prepending the name of the class to the name of the generic. The preprocessor should first determine which operations are actually required, thus preventing the unnecessary addition of overhead to both the compilation and the resulting application.

Even though the generic-based approach provides substantially more error checking at compile time, the address-based approach works best given Ada 83's inability to model inheritance. Defining a single object type in this manner is somewhat analogous to defining a root object class from which all persistent-capable objects can be derived. An ideal approach would be similar to the C++ binding approach incorporating root class `Pobject` (1:98). Unfortunately, inheritance will need to be modeled (or simulated) in the underlying ODBMS implementing the Ada binding. As a result, defining object parameters as address pointers allows the ODBMS a greater degree of flexibility in how it represents inheritance and objects in general.

Ada's inability to model inheritance presents an additional hindrance to the generic-based approach. To access an ODMG-93 construct defined as a generic, that construct would have to be instantiated for every object class in the database rather than simply for the root class. Consequently, the class of an object retrieved through a relationship traversal, for example, would need to be determined before it could be manipulated with the appropriate generic. This situation violates the principles of reusability and maintainability fundamental to the object-oriented paradigm. Introducing a new class in the inheritance hierarchy would require including a test for that object class in every desired ODMG-93 construct, resulting in an application that would be difficult to maintain. Therefore, the address-based approach is selected as the overall approach to object definition. Each package in the Ada binding defines its own type `Object` as a subtype of `System.Address`.

The remainder of this chapter presents the Ada language binding to ODMG-93 Object Definition Language (ODL), Object Manipulation Language (OML), and Object Query Language (OQL). Because this binding is best illustrated in the context of an object model (19:38–43), a simple one is proposed in Figure 3.4. Class `Person` is the base class, and classes `Student` and `Faculty` both inherit attributes from `Person` while adding additional attributes. `Spouse` is a one-to-one relationship between two people, while `Advisor` and `Advisees` are one-to-one and one-to-many traversal paths between student and faculty objects, respectively. A `Person` can have zero or one spouse, a `Faculty` can have zero or more advisees, and a `Student` must have exactly one advisor.

3.4 Ada ODL

After selecting the address-based approach for designing the interface, the Ada language binding for the remaining ODMG-93 constructs follows naturally. Like any abstract data type implemented in Ada, object classes are represented as packages.

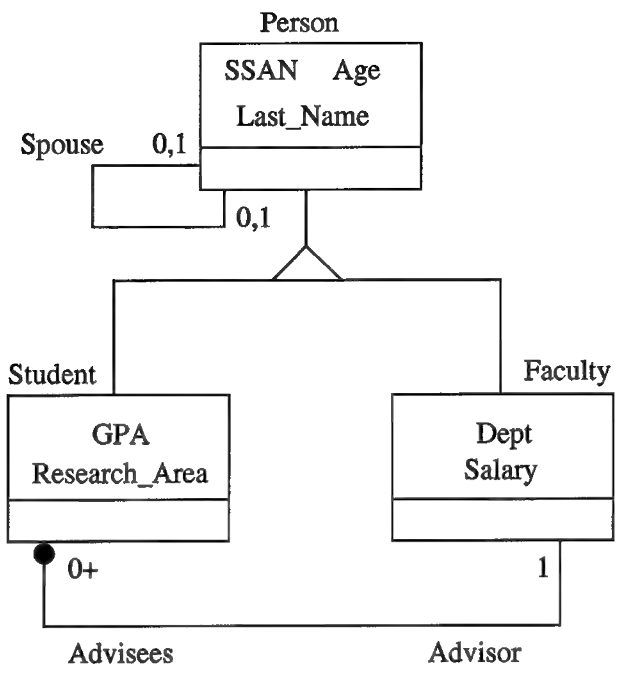

Figure 3.4 Example Object Model

3.4.1 Attributes and Methods. Figure 3.5 shows a typical object class definition using the address-based approach. Type **`Object`** is the aforementioned **`System.Address`**, and **`Extent`** allows the user to specify where all instances will be located. Attributes are defined using a record in the private section of the package, preventing direct access by any calling routines. As a result, **`Set_Value`** and **`Get_Value`** routines are required to manipulate object attributes. While these could be defined by attribute, such as **`Get_SSAN`** and **`Get_Age`**, a less cumbersome technique is to define them by type. This requires potentially fewer accessor routines that could be easily defined by the ODL preprocessor rather than by the user.

Methods are represented by defining procedures and functions to be executed for an instance; method **`Birthday`** is included in our example. Attribute initial values are defined by the user in

```
with System, Database;
package Person is

  Extent : constant string := "Persons";

  subtype Object  is System.Address;
  type Attributes is private;
  type Reference  is access Attributes;

  function Create(A_Database : in Database.Object;
                  Last_Name  : in String  := "";
                  SSAN       : in Integer := 0;
                  Age        : in Integer := 0) return Object;

  procedure Delete(A_Database : in Database.Object;
                   A_Person   : Object);

  procedure Set_Value(A_Person  : in out Object;
                      Attribute : in     String;
                      Value     : in     Integer);

  procedure Set_Value(A_Person  : in out Object;
                      Attribute : in     String;
                      Value     : in     String);

  function Get_Value(A_Person  : in Object;
                     Attribute : in String) return Integer;

  function Get_Value(A_Person  : in Object;
                     Attribute : in String) return String;

  procedure Birthday(Instance : in out Object);

private

  type Attributes is
    record
      Last_Name : String(1..30);
      SSAN      : Integer;
      Age       : Integer;
      Spouse    : Person.Reference inverse Person.Spouse;
    end record;

end Person;
```

Figure 3.5 Example Object Class Definition

the `Create` function specification. Object creation and deletion is discussed in Section 3.5.1, and attribute `Spouse` is a one-to-one relationship, discussed in Section 3.4.2.

3.4.2 Relationships. Relationships in an object database provide a facility for maintaining data integrity in an association between objects with minimal effort on behalf of the user. ODMG-93 refers to a given relationship between objects not by an assigned name, but by named *traversal paths* (1:23). Data integrity is then maintained by declaring inverse traversal paths.

Defining relationships and object inheritance in Ada ODL can be accomplished with the syntax proposed by Moyers (14:B-1–B-4). Figure 3.6 illustrates the attribute record definitions for traversal paths `Advisor` and `Advisees` presented in Figure 3.4. Collections (discussed in section 3.5.3) are used to implement one-to-many and many-to-many relationships. One-to-one relationships are implemented as pointers, rather than copies of objects, to preserve data integrity. The inverse traversal paths imply that adding a student to a faculty member's set of advisees will automatically update that student's advisor to the given faculty instance. Because keyword `inverse` is not part of standard Ada, it must be removed by the ODL preprocessor prior to compilation. Additionally, circular references may be incurred by having two object class specifications depend on each other to declare inverse traversal paths, such as `Advisor` and `Advisees`. The corresponding `with` statements must also be removed by the preprocessor.

Additionally, the clause `new Person with` found in each package of Figure 3.6 is not standard Ada and is used only to communicate inheritance to the ODL preprocessor. These clauses must also be removed prior to compilation, and therefore a transient instantiation of record `Attributes` for an inherited object will return a data structure that includes only non-inherited attributes. In this regard, the Ada binding does not completely conform to ODMG-93, whose goal is to allow the

```
with Person, Faculty;
package Student is
  ...
private
  type Attributes is new Person with
    record
      Research_Area : String;
      GPA           : Float;
      Advisor       : Faculty.Reference inverse Faculty.Advisees;
    end record;
end Student;


with Person, Student, Set;
package Faculty is
  ...
private
  package Student_Set is new Set("Student");

  type Attributes is new Person with
    record
      Dept     : String;
      Salary   : Float;
      Advisees : Student_Set.Object inverse Student.Advisor;
    end record;
end Faculty;
```

Figure 3.6 Example Relationship and Inheritance Definition

user to manipulate both persistent and transient instances of objects. Note that this shortcoming is a result of Ada 83's inability to model inheritance, which forces the user to specify inheritance in non-Ada terms so that it may be modeled in the underlying ODBMS.

3.5 Ada OML

This section identifies the Ada OML binding to ODMG-93. In the following subsections each construct is reviewed and mapped to Ada.

3.5.1 Object Creation and Deletion. Consistent with the other ODMG-93 language bindings, Ada programmers must have the ability to manipulate both persistent and transient instances of objects. Overloading the `Create` and `Delete` operations can accomplish this.

At a minimum, the function specification for the persistent version of `Create` must be provided by the user, as this is how attribute initial values are communicated to the ODL preprocessor (Section 3.4.1). This specification must include a parameter of type `Database.Object` to inform the ODBMS to create a persistent object and to store it in the given file. Because each ODBMS implements object creation differently, the body of `Create` will be uniquely defined for a particular ODBMS. The `Delete` procedure is implemented similarly.

If the user also wishes to manipulate transient instances, he or she is responsible for defining one or more additional `Create` and `Delete` routines without including the `Database.Object` parameter. These routines will simply be standard Ada procedures and functions based on the user's needs. Since objects are represented using subtype `System.Address`, the user is responsible for returning a converted object reference during creation.

3.5.2 Properties. Attributes and relationships are the two types of properties defining an object's state. A persistent name is an additional object property that has no effect on an object's state; it is used for nonassociative access of a single persistent object and exists primarily for the benefit of the programmer (1:17–18). By assigning an object a persistent name, a programmer can later lookup the object without having to devise a complex query.

3.5.2.1 Attributes. ODMG-93 states that the following built-in operations must be defined for all attributes (1:22):

$$set\_value(new\_value : Literal)$$
$$get\_value() \rightarrow existing\_value : Literal$$

Each of the attributes in our example object model of Figure 3.4 is made accessible by defining the `Set_Value` and `Get_Value` routines discussed in Section 3.4.1. As previously stated, the user may manipulate only noninherited attributes of transient instances.

3.5.2.2 Relationships. ODMG-93 defines the following operations for manipulating relationships (1:24):

<u>One-to-one</u>

$$create(o1 : Object, o2 : Object)$$
$$delete()$$
$$traverse(from : Object) \rightarrow to : Object$$

<u>One-to-many</u>

$$create(o1 : Object, s : Set{<}Object{>})$$
$$delete()$$
$$add\_one\_to\_one(o1 : Object, o2 : Object)$$

$$remove\_one\_to\_one(o1 : Object, o2 : Object)$$
$$traverse(from : Object) \rightarrow to : Set{<}Object{>}$$

<u>Many-to-many</u>

$$delete()$$
$$add\_one\_to\_one(o1 : Object, o2 : Object)$$
$$remove\_one\_to\_one(o1 : Object, o2 : Object)$$
$$add\_one\_to\_many(o1 : Object, s : Set{<}Object{>})$$
$$remove\_one\_to\_many(o1 : Object, s : Set{<}Object{>})$$
$$remove\_all\_from(o1 : Object)$$

Note that the **`create`** and **`traverse`** operations are not explicitly defined on many-to-many relationships, as a many-to-many relationship is actually implemented as two component one-to-many relationships. Therefore, the user invokes the corresponding component operations (1:24).

Each of the above constructs maps easily into Ada using the address-based approach. The resulting package **`Relationships`** is shown in Appendix A.8. An additional procedure **`Initialize`** is included in this package, as an ODBMS implementation may require special relationship initialization processing. This procedure must be invoked before any other relationship operations.

3.5.2.3 Persistent Names. ODMG-93 discusses the importance of persistent names but does not explicitly identify the operations for their manipulation. However, in the C++ language binding implementation, the constructs appear as follows (1:108–110):

$$name(o : Object, theName : String)$$
$$lookup(theName : String) \rightarrow o : Object$$

These routines also map directly and easily into the binding. The resulting `Name_Object` and `Lookup_Object` operations are placed in package `Database`, discussed in Section 3.5.5.

3.5.3 Collections. A collection is simply a grouping of objects of a particular class. One type of collection, `Set`, was introduced in defining relationships. An overview of the behavior of each type is as follows:

- `Set`: an unordered collection disallowing duplicates
- `Bag`: an unordered collection allowing duplicates
- `List`: an ordered collection allowing duplicates
- `Array`: a contiguous block of ordered memory locations with a finite upper bound, each cell of which stores a single object, allowing duplicates

Collections are best discussed in the context of inheritance, with class `Collection` as the superclass and each of its four types the unique subclasses. This structure is reflected in Figure 3.7. All of the operations of each class are identified in the object database standard (1:26–35). Operation `Select` must be renamed `Select_Subcollection` to avoid incorrect usage of an Ada reserved word. Similarly, package `Array` is renamed `Array_Type`.

Ada 83 cannot model inheritance, so binding the operations in class `Collection` to Ada necessitated repeating them in each of the four individual subclass packages, as shown in appendices A.3, A.4, A.5, and A.6 for package specifications `Set`, `Bag`, `List`, and `Array_Type` respectively. Ideally, the instantiation type should be the type of the collected object; however, this makes little sense in the address-based approach, as all object types are subtypes of `System.Address`. In fact,

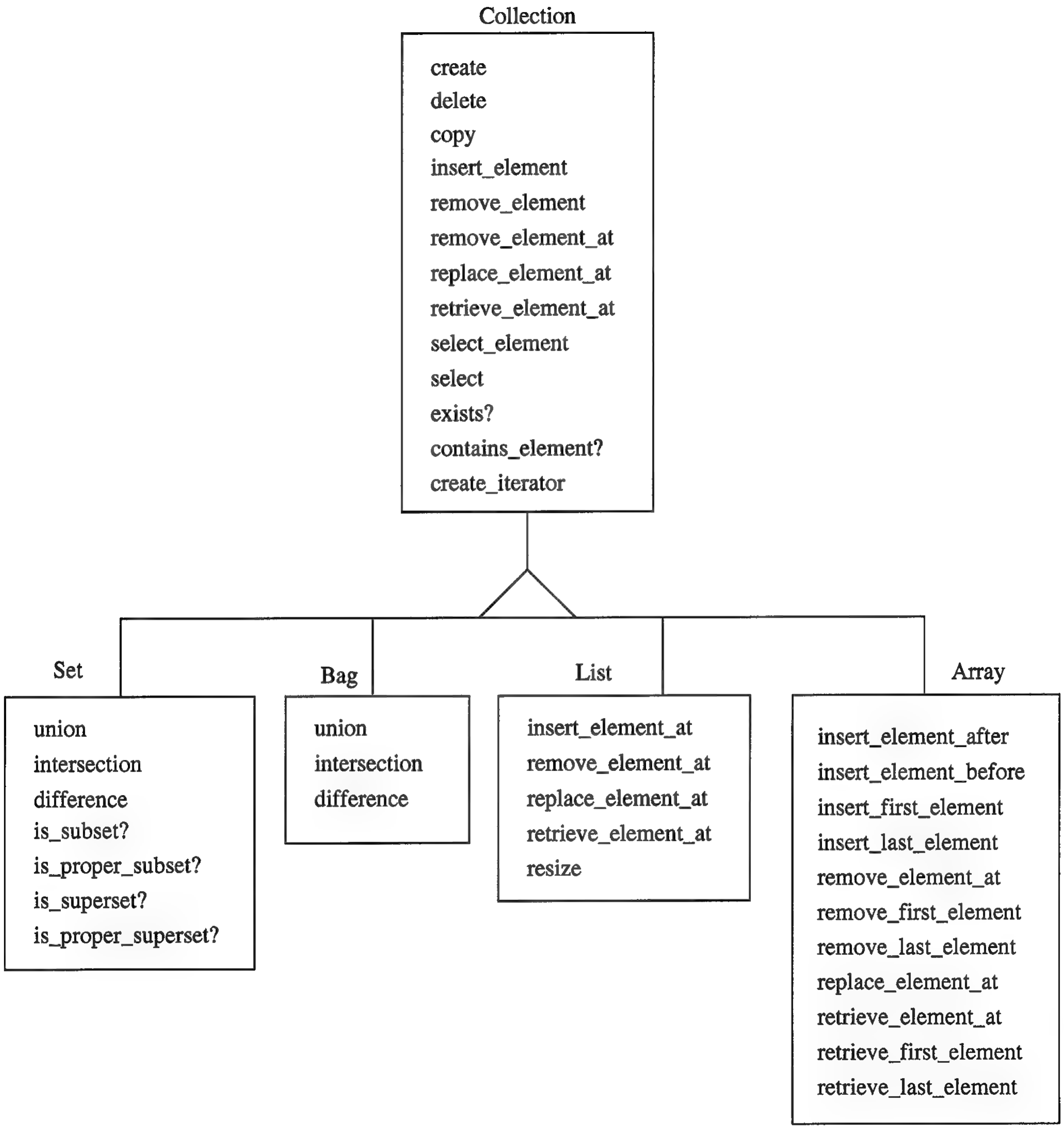

Figure 3.7 Operations for Class `Collection` and its Subclasses

defining each collection with a generic instantiation type would require type conversions between the instantiation type and the object type, both of which are **`System.Address`**. Therefore, to provide each underlying ODBMS with additional information regarding the class of a collected object, each is implemented as a generic with a string instantiation parameter representing its class. Figure 3.6 shows a typical instantiation in defining the one-to-many relationship traversal path **`Advisees`**.

Once a collection is created and initialized, a facility must exist for traversing the elements, and package **`Iterator`** provides this capability. The operations available for iterators include the following (1:29–30):

$$\begin{gathered} first() \rightarrow element : Object \\ last() \rightarrow element : Object \\ next() \rightarrow element : Object \\ more() \rightarrow b : Boolean \\ reset() \\ delete() \end{gathered}$$

The corresponding Ada package specification is shown in Appendix A.7. The same string instantiation parameter used for collections is also used here. Procedures **`First`**, **`Last`**, and **`Next`** position the iterator and return the object located at that position as an **`out`** parameter. If an object does not exist at this new position, a null value (constant **`System.NO_ADDR`**) is returned. Procedure **`Reset`** places the iterator at the beginning of the collection without returning the corresponding object. Note that a **`Create`** function does not exist in this package, as iterators are created in the collection package defining the collection to be traversed.

3.5.4 Transactions. Package **`Transaction`** encapsulates all transaction operations, which are presented in ODMG-93 as follows (1:40–42):

$$begin() \rightarrow t : Transaction$$
$$commit()$$
$$abort()$$
$$checkpoint()$$
$$abort\_to\_top\_level()$$

The `begin` operation creates and starts a transaction. Operation `commit` saves all changes made during the current transaction; `abort` causes all changes since the corresponding `begin` to be undone. Both delete the current transaction instance, and both apply to persistent objects only. Because transactions may be nested in the ODMG-93 model, the `commit` operation is relative to the transaction at the highest nesting level. Invoking `checkpoint` causes all persistent changes to be saved without deleting the transaction instance. Finally, `abort_to_top_level` aborts all outstanding transactions to the highest level.

Package `Transaction` is included in Appendix A.2. Similar to the other high-level type definitions in the address-based approach, `Object` is defined as a subtype of `System.Address`. Since `abort` is an Ada keyword, this routine is named `Abort_Txn`. For consistency, the commit operation is renamed `Commit_Txn`.

3.5.5 Database Operations. Operations on databases include the following (1:42):

$$open(dbName : String)$$
$$close()$$
$$contains\_object?(o : Object) \rightarrow b : Boolean$$
$$name\_object(o : Object, theName : String)$$
$$lookup\_object(theName : String) \rightarrow o : Object$$

The `open` and `close` operations manipulate database files in the expected manner, and an object's existence in a database can be determined by invoking `contains_object`. The `name_object` and `lookup_object` operations manipulate persistent naming discussed in Section 3.5.2.3.

Package `Datbase` is presented in Appendix A.1. Type `Database.Object` is also defined as a subtype of `System.Address`. Global variable `Database.Current` is initialized in each call to `Open_Database`. With the exception of object creation and deletion, all constructs requiring a database file are defined with the default database parameter `Database.Current`. This allows the user to manipulate persistent objects and collections without specifying the same database file for each one; however, he or she still has the flexibility to pass a different database file and avoid manipulating the default value. Object creation and deletion cannot be defined in this manner, as the `Database.Object` parameter determines the lifetime of the object. Constructs such as persistent name manipulation, for example, can operate only on persistent objects, so a default database parameter is allowable and provides an extra degree of convenience for the user.

Two additional operations, `Initialize_Interface` and `Stop_Interface`, are included to perform any initialization and finalization processing required by the underlying ODBMS. The user is required to invoke these as the first and last routines, respectively, in an Ada database application. Depending on the needs of the system, their corresponding implementations may be nothing more than a no-op.

3.6 Ada OQL

Besides the basic advantage of persistence itself, a DBMS's query facility represents the heart of its functionality. The ability to create and store objects, either individually or from within collections, is of limited value if those objects cannot be retrieved in an organized manner.

3.6.1 ODMG-93 OQL.

The query facility of ODMG-93, OQL, describes this capability for an object database (1:66–81). Its syntax is similar to the SQL `select-from-where` clauses for relational databases (10:97–119). The result type of the query output is determined by the `select` clause.

Figure 3.8 shows several example OQL queries. The first query returns the set of all instances of class `Person` who are more than 21 years old. The second query returns the single `Student` instance whose name is "Jones," or raises an exception if more than one student shares this name (1:77). The third query returns a bag of float values representing the salaries of every `Faculty` instance. The fourth query returns a set of ordered pairs showing the correlation between the `Age` and `Salary` attributes for a subset of faculty members. Finally, the last two queries both return the set of all `Student` instances.

These examples illustrate several important points. First, the `element` keyword returns the single element satisfying the given query expression. Otherwise, `select distinct` is used in returning a set collection subclass while `select` is used in returning a bag. The only exception is a query selecting the elements of an extent. Such queries (examples 1, 5, and 6 of Figure 3.8) always return sets, since by definition the elements of an extent are unique in that each one has a unique object identifier (OID). In addition, dynamic structures can be created "on the fly," as shown in the fourth example.

```
1. select x from x in Persons where x.age > 21

2. element(select x from x in Students where x.last_name =
   "Jones")

3. select x.salary from x in Faculties

4. select distinct struct(age :  x.age, salary:  x.salary)
   from x in Faculties where x.age > 40

5. select x from x in Students

6. Students
```

Figure 3.8 Example ODMG-93 Queries

```
function Query(Expression : in String;
               A_Database : in Database.Object := Current)
  return System.Address;
```

Figure 3.9 Ada Query Function

Obviously, OQL is a powerful facility, so much so that the ODMG-93 authors realize many ODBMS vendors will struggle with its implementation:

> Although the language allows arbitrary program-defined functions to be invoked within the body of the query, it will require a fairly sophisticated runtime QL interpreter to locate, bind, and execute the methods that implement these functions. We expect that many systems may restrict this capability (16:120).

3.6.2 Ada Binding to OQL. The query function is placed in package `Database`, and is shown in Figure 3.9. Because it returns a value of type `System.Address`, this value may be either a collection or an object.

Two language binding alternatives exist for implementing a query language: *loosely coupled* and *tightly coupled.* In the tightly coupled approach, the query keywords and expressions are incorporated within the language, allowing the parsing and optimizing of queries to occur at compile

```
1. Over21 :  Set.Object := Database.Query(
           "select X from X in Persons where X.Age > 21");

2. Jones :  Student.Object := Database.Query(
           "element(select X from X in Students
           where X.Last_Name = 'Jones')");
```

Figure 3.10 Example Ada OQL Queries

time. The loosely coupled approach represents queries with strings that are parsed, optimized, and executed at runtime. Due to its relative simplicity, the loosely coupled approach is chosen for the Ada binding. The primary disadvantage of this approach is the additional runtime required to optimize queries that can be optimized at compile time in the tightly coupled approach. Additionally, syntax errors in queries that are not well-formed are discovered at runtime, rather than at compile time, possibly halting execution of an otherwise syntactically correct program.

As a result of selecting the loosely coupled approach, the first two queries in Figure 3.8 may be represented in the Ada binding with the calls shown in Figure 3.10. All predicates incorporate standard Ada syntax. Although direct attribute access such as `X.Age` is not allowed in Ada ODL, it is allowed in OQL to simplify predicate expressions. For further simplicity, strings may be specified within the string expression itself by using single quotes as delimiters.

3.7 Summary

Two methods for designing an Ada binding to ODMG-93 were examined in this chapter: the address-based and generic-based approaches. The address-based approach was chosen, as it more easily allows an underlying ODBMS to offset Ada 83's inability to model inheritance. Ada bindings to ODMG-93 ODL, OML, and OQL were then identified. The next chapter begins testing the binding by developing ODL implementations for Itasca and ObjectStore.

IV. Approach Selection and ODL Implementation

4.1 Overview

This chapter begins by identifying two contrasting approaches to implementing the Ada binding for an ODBMS product. A subset of Ada Object Definition Language (ODL) is then implemented for Itasca and ObjectStore. All package modifications and additional source code output required by each ODL preprocessor are discussed along with a summary of the results.

4.2 Approach Selection

Both Itasca and ObjectStore have C application program interfaces (APIs) accessing their functionality, and the Sun Ada compiler's **pragma** directives allow the Ada binding to be implemented by linking to each vendor's C library. Before implementation of the Ada binding presented in Chapter III can begin, however, an approach must be selected. Two methods are available: *indirect* and *direct.*

4.2.1 Indirect Method. The indirect method takes each of the ODBMS constructs used in implementing the Ada binding and defines an Ada counterpart in the form of a function, procedure or generic package, thus producing a one-to-one functional mapping from C to Ada. The Ada binding calls one or more of the corresponding routines in the Ada/ODBMS package, which in turn creates any necessary parallel data types to translate parameters into the format required by C. This approach allows the binding implementation to be written completely in Ada, while the direct interfacing to the actual ODBMS routines written in C occurs one level below the binding.

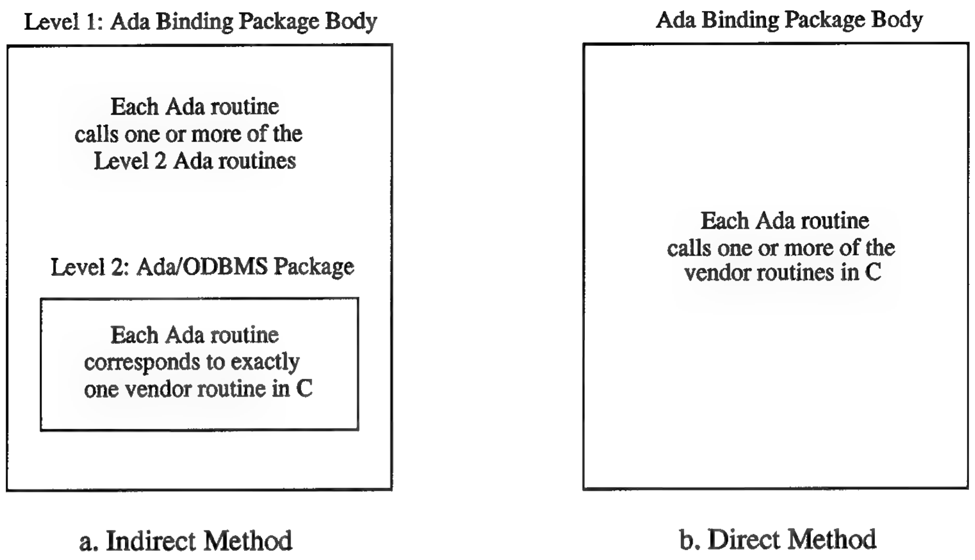
Level 1: Ada Binding Package Body
Each Ada routine calls one or more of the Level 2 Ada routines
Level 2: Ada/ODBMS Package
Each Ada routine corresponds to exactly one vendor routine in C
Ada Binding Package Body
Each Ada routine calls one or more of the vendor routines in C
a. Indirect Method
b. Direct Method

Figure 4.1 Alternatives for the Ada Binding Interface

4.2.2 Direct Method. The direct method uses Ada's **pragma** directives to access the C routines of the API directly. This method involves first determining the routines to be invoked and the parallel data types to be created, then invoking those routines with the corresponding parameters, all in the package body of the Ada binding.

These contrasting methods are represented visually in Figure 4.1. Figure 4.1a shows the extra interface layer accounted for by the indirect method. This extra layer is nonexistent in the direct method implementation, as seen in Figure 4.1b.

4.3 Comparison of Alternatives

Each implementation method has its own set of advantages and disadvantages, summarized in Tables 4.1 and 4.2.

Table 4.1 shows the indirect method has numerous advantages and only one disadvantage not common to the direct method: implementing an additional specification and body. This disadvan-

Table 4.1 Advantages and Disadvantages of the Indirect Method

Advantages	**Disadvantages**
Simpler and shorter Ada binding implementation, since Ada routines call other Ada routines	Must implement an additional specification and body to provide the one-to-one functional mapping
No **pragma** directives or non-Ada type conversions are needed in the Ada binding	**Pragma** directives and type conversions are still needed at Ada/ODBMS level, if a vendor-supplied interface is not available
Any new vendor Ada interfaces can be easily incorporated by replacing them in the Ada/ODBMS package	
Vendor modifications to a C interface routine involves changing only one occurrence of that routine	

Table 4.2 Advantages and Disadvantages of the Direct Method

Advantages	**Disadvantages**
Only one package specification and body is needed to implement the interface	Can be difficult to trace vendor modifications to an interface routine, since it will likely be referenced in more than one location
No need to worry about type conversions, since data types are not referenced through an additional interface layer.	**Pragma** directives and type conversions are still needed if a vendor-supplied interface is not available

tage is heavily outweighed by the modularity of the indirect method, which ultimately makes this method more reusable and maintainable. Coincidentally, this is the primary disadvantage of the direct method, since tracing modifications to vendor routines becomes increasingly cumbersome as the binding functionality expands.

As a result, the indirect method is selected to implement the Ada binding for Itasca and ObjectStore in the remainder of this thesis. For ObjectStore, this functional mapping already exists in the Ada/ObjectStore prototype interface developed by Object Design, Inc. (18) and extended by Li Chou (4). Itasca had no such mapping, so its implementation required first producing package `Itasca`. The constructs in each of these Ada/ODBMS interfaces are discussed as they are presented in the Ada binding implementations of this thesis.

4.4 ODL Implementation

The remainder of this chapter discusses implementing Ada ODL for Itasca and ObjectStore. The object model of Figure 3.4 used in discussing the binding is also used to test its implementation. The unique portions of the package specifications defining object classes **`Person`**, **`Student`**, and **`Faculty`** (omitting the **`Create`**, **`Delete`**, and attribute manipulation routines) are shown in Figures 4.2, 4.3, and 4.4 respectively. Due to time constraints, methods are not implemented in this thesis. Likewise, many-to-many relationships are not implemented, although the example one-to-one and one-to-many traversal paths **`Student.Advisor`** and **`Faculty.Advisees`** are.

In examining the package bodies of Figures 4.2 through 4.4, it is apparent that each ODL preprocessor must perform the following tasks at a minimum:

- Remove circular dependencies between packages

```
with System, Database;

package Person is

  Extent : constant string := "Persons";
  ...

private

  subtype Attribute_String is String(1..30);

  type Attributes is record
    Last_Name : Attribute_String;
    SSAN      : Integer;
    Age       : Integer;
    Spouse    : Person.Reference inverse Person.Spouse;
  end record;

end Person;
```

Figure 4.2 Abbreviated **Person** Class Definition

```
with System, Database;
with Person, Faculty;

package Student is

  Extent : constant string := "Students";
  ...

private

  subtype Attribute_String is String(1..30);

  type Attributes is new Person.Attributes with
    record
      GPA           : Float;
      Research_Area : Attribute_String;
      Advisor       : Faculty.Reference inverse Faculty.Advisees;
    end record;

end Student;
```

Figure 4.3 Abbreviated **Student** Class Definition

```
with System, Set, Database;
with Person, Student;

package Faculty is

  Extent : constant string := "Faculties";
  ...

private

  package Student_Set is new Set("Student");
  subtype Attribute_String is String(1..30);

  type Attributes is new Person.Attributes with
    record
      Salary   : Float;
      Dept     : Attribute_String;
      Advisees : Student_Set.Object inverse Student.Advisor;
    end record;

end Faculty;
```

Figure 4.4 Abbreviated Faculty Class Definition

- Create the database schema
  - Create extents for each class
  - Determine attribute representation and remove inheritance clauses
  - Determine relationship representation and remove keyword `inverse`
- Store the resulting schema in the appropriate database file

Each ODL preprocessor must also perform additional tasks relevant to its particular ODBMS implementation, all of which are discussed in the following sections.

4.4.1 Itasca ODL Implementation. Removing circular dependencies is trivial and involves simply "commenting out" the corresponding `with` statements and any references to those packages. In our example, the statement `with Student` must be removed from package `Faculty` along with the declaration of traversal path `Advisees`; a similar modification is required for package `Student`. Non-ada keywords such as the inheritance clause `is new <X>.Attributes with` must also be removed. Next, the Itasca ODL preprocessor translates the user's schema information into the C code creating these classes in the database. The C code fragment to create our example classes (representing only attributes, and not relationships) is shown in Figure 4.5. Although it is not shown in the code illustration, the `Iitasca_init` command creates the database file in which to store the resulting database schema; its file name is scanned from the application.

Figure 4.5 illustrates several important points. First, each attribute is created by invoking `IUmake_attribute` with the appropriate name and type. The class is then created by calling `Imake_class`. Derived classes may specify inheritance using the `SUPERCLASSES` keyword. Extents are automatically created in Itasca and require no special output from the preprocessor.

```
/* class Person */

  attr1 = IUmake_attribute("SSAN", DOMAIN, "integer", END_ARGS);
  attr2 = IUmake_attribute("Last_Name", DOMAIN, "string", END_ARGS);
  IUappend(attr1, attr2);
  attr2 = IUmake_attribute("Age", DOMAIN, "integer", END_ARGS);
  IUappend(attr1, attr2);

  if (Imake_class(&uid, "Person", ATTRIBUTES, attr1, END_ARGS))
    printf("  Schema Error: %s\n", ierrstring);
  IUfree_generic(attr1, TYPE_VAL_LIST);

/* class Student */

  attr1 = IUmake_attribute("Research_Area", DOMAIN, "string", END_ARGS);
  attr2 = IUmake_attribute("GPA", DOMAIN, "float", END_ARGS);
  IUappend(attr1, attr2);

  if (Imake_class(&uid, "Student", SUPERCLASSES, "Person",
              ATTRIBUTES, attr1, END_ARGS))
    printf("  Schema Error: %s\n", ierrstring);
  IUfree_generic(attr1, TYPE_VAL_LIST);

/* class Faculty */

  attr1 = IUmake_attribute("Dept", DOMAIN, "string", END_ARGS);
  attr2 = IUmake_attribute("Salary", DOMAIN, "float", END_ARGS);
  IUappend(attr1, attr2);

  if (Imake_class(&uid, "Faculty", SUPERCLASSES, "Person",
              ATTRIBUTES, attr1, END_ARGS))
    printf("  Schema Error: %s\n", ierrstring);
  IUfree_generic(attr1, TYPE_VAL_LIST);
```

Figure 4.5   Preprocessor Output Creating Itasca Object Classes and Attributes

```
Ichange_attribute(&uid, "Person", "Spouse", FALSE,
     ATTR_DOMAIN, "Person", END_ARGS);
```

Figure 4.6 Itasca Creation of Relationship Spouse

Relationships are defined in Itasca by specifying attributes referring to a single object (one-to-one) or to sets of objects (one-to-many, many-to-many). To incorporate relationship Spouse, the subroutine call of Figure 4.6 must be added after class Person is created. This operation adds the relationship as a pointer to any object of the previously created domain Person. Note that the relationship must be specified using an attribute change, rather than the IUmake_attribute command used for attributes Research_Area and GPA, as domain Person is undefined until the call to Imake_class executes.

The same type of circular referencing exists for the Advisor/Advisees relationship: a student's Advisor is of type Faculty, and cannot be referenced until the class is created, while a faculty member's Advisees are of type (set-of Student), and require a complete definition of class Student. The solution is to create one of the two classes without the relationship, completely create the other class, then modify the first one to include the omitted relationship.

Figure 4.7 illustrates this process. Class Student is first created without the Advisor attribute, as shown in Figure 4.6. Class Faculty is then created to include attribute Advisees, making it completely defined. The Ichange_attribute command then adds the Advisor attribute to class Student, completing its definition.

An additional note involves relationship inverses. Because Itasca has no facility for determining which relationships are inverses of each other, this information must be retrieved by the ODL preprocessor and stored for later processing. The Itasca ODL implementation uses an exter-

```
/* class Faculty */
  ...
  attr2 = IUmake_attribute("Advisees", DOMAIN, "(set-of Student)", END_ARGS);
  IUappend(attr1, attr2);

  if (Imake_class(&uid, "Faculty", SUPERCLASSES, "Person",
              ATTRIBUTES, attr1, END_ARGS))
    printf("  Schema Error: %s\n", ierrstring);
  IUfree_generic(attr1, TYPE_VAL_LIST);

/* Modify class Student */

  Ichange_attribute(&uid, "Student", "Advisor", FALSE,
     ATTR_DOMAIN, "Faculty", END_ARGS);
```

Figure 4.7 Creation of Itasca Relationships For Classes **Student** and **Faculty**

```
Faculty.Advisees 1:m
Student.Advisor 1:1
Person.Spouse 1:1
Person.Spouse 1:1
```

Figure 4.8 Itasca Relationship Inverse File

nal file **inverse.dat**, depicted in Figure 4.8, to store relationship inverse pairs along with their multiplicities. This information is required for OML implementation, discussed in Chapter V.

4.4.2 ObjectStore ODL Implementation.

The implementation of ObjectStore ODL is more complex than its Itasca counterpart, primarily because a facility does not exist for specifying inheritance between classes. As a result, a technique for simulating inheritance must be chosen.

4.4.2.1 Inheritance Simulation in ObjectStore.

Figure 4.9 shows the common method of embedding an ancestor object within an inherited object. Accessing the **Person** attributes of a **Student** instance using this technique, however, requires first accessing the student

```
                  type Person is record
                    SSAN : Integer;
                    Age : Integer;
                    Last_Name : String;
                  end record;


type Student is record                    type Faculty is record
  P : Person;                               P : Person;
  GPA : Float;                              Dept : String;
  School : String;                          Salary : Float;
end record;                               end record;
```

Figure 4.9 Embedded Object Method of Simulating Inheritance

instance. This presents a problem in the ObjectStore implementation when the user wishes to query extent `Persons`. Because the requested class is embedded, it does not appear in the extent.

An immediate solution to this problem is to create an additional instance of the ancestor class and place it in the ancestor extent. Now, however, two distinct objects (that actually refer to the same object) exist in the database, and updating an attribute of one will not update the corresponding attribute of the other. This situation results in a breach of data integrity.

A better solution is to embed not an ancestor object but a *pointer* to an ancestor object. This way, only one copy of the ancestor attributes exists in the database. Queries can now be performed on both extents and data integrity is perserved. Additionally, the inherited attributes can be accessed by simply dereferencing the ancestor pointer, a process that can be repeated to an arbitrary number of levels. The object model representing this technique is shown in Figure 4.10, the model chosen for the ObjectStore ODL implementation.

4.4.2.2 Implementing ObjectStore ODL. The technique for removing circular package dependencies is identical to that used for Itasca. Because ObjectStore does not automatically

```
type Person is record
   SSAN : Integer;
   Age : Integer;
   Last_Name : String;
end record;
```

```
type Student is record                    type Student is record
   Ancestor : Person_Ptr;                    Ancestor : Person_Ptr;
   GPA : Float;                              Salary : Float;
   School : String;                          Dept : String;
end record;                               end record;
```

Figure 4.10 ObjectStore ODL Object Model

place new instances in an extent, a corresponding collection must be created for each class. This is done during object creation, discussed in Chapter V. The ObjectStore ODL preprocessor then defines attributes by creating a separate database schema file using an Ada-like syntax and the Ada/ObjectStore header file `os_ada.hh`. This header file is a set of macros that automatically generate the C parallel data types based on the attributes specified by the user.

Relationships present an additional challenge to the ObjectStore ODL implementation. Because Ada/ObjectStore is a prototype interface with limited functionality, it does not have the ability to store a pointer to a set of objects within a persistent object. However, Ada/ObjectStore can store integer types, and since an Ada integer type is the same length (32 bits) as an address or pointer type in C, relationships can be represented as type `Integer` and converted to their actual object and collection addresses during manipulation.

The resulting ObjectStore schema file produced by the ODL preprocessor for our example object model is depicted in Figure 4.11. Attribute `Ancestor` is defined as a pointer to the immediate superclass of an object and appears as expected in the definitions of `Student` and `Faculty`. String

```
#include "os_ada.hh"

DEFADAREC(Person)
DEFVECFIELD(Last_Name,CHAR,30)
DEFIELD(SSAN,INTEGER)
DEFIELD(Age,INTEGER)
DEFIELD(Spouse,INTEGER)
ENDADAREC(Person)

DEFADAREC(Student)
DEFIELD(Ancestor,ACCESS(Person))
DEFIELD(GPA,FLOAT)
DEFVECFIELD(Research_Area,CHAR,30)
DEFIELD(Advisor,INTEGER)
ENDADAREC(Student)

DEFADAREC(Faculty)
DEFIELD(Ancestor,ACCESS(Person))
DEFIELD(Salary,FLOAT)
DEFVECFIELD(Dept,CHAR,30)
DEFIELD(Advisees,INTEGER)
ENDADAREC(Faculty)
```

Figure 4.11 Ada/ObjectStore Schema File

attributes are defined with the length specified by the user; float and integer attributes translate directly.

Finally, the ODL preprocessor is required to change the attribute record definitions in each object class package to reflect the schema shown in Figure 4.11. This is because attribute and relationship manipulation in Ada/ObjectStore involves updating the field values of these records, discussed further in Chapter V. The Itasca ODL preprocessor has no such requirement, as attributes and relationships are manipulated entirely in the ODBMS implementation language, and not in Ada.

ObjectStore must also store relationship inverse and multiplicity information in an external data file. The same format used by Itasca is used here.

4.5 ODL Implementation Results

Clearly, the Itasca implementation of Ada ODL is more elegant and straightforward due primarily to Itasca's ability to specify inheritance. Additionally, parallel data types are not required because attributes and relationships are represented entirely in the ODBMS implementation language, which is Lisp[1], and not in Ada. Finally, Itasca's **`set-of`** keyword allows relationships to be defined directly and intuitively as references to objects and to sets of objects, while relationships in ObjectStore must be defined with an integer attribute to act as a placeholder for an address.

4.6 Summary

Two contrasting Ada binding implementation approaches were examined in this chapter, and because of its increased maintainability and expandability, the indirect method is chosen over the direct method. Ada ODL was then implemented for Itasca and ObjectStore using a simple object model, and the Itasca implementation proved to be more straightforward than the ObjectStore version. The next chapter further tests the Ada binding by discussing OML implementation.

[1]Although the Itasca C API is used to access its functionality, the implementation language is Common Lisp with the C API used as an additional interface layer (21:1).

V. OML Implementation

5.1 Overview

This chapter discusses implementation of a subset of Ada Object Manipulation Language (OML) operations for Itasca and ObjectStore. The results of each are then summarized and compared.

5.2 Introduction

Implemented Ada OML operations are reviewed here in the order they were introduced in Chapter III. While the user is allowed to manipulate both transient and persistent instances in the Ada binding, this chapter is devoted primarily to the manipulation of persistent objects. Special considerations for transient object manipulation are included prior to the discussion of Ada OML implementation results.

5.3 Object Creation and Deletion

5.3.1 Itasca. Once the schema is created by the ODL preprocessor, creating an object in an Itasca database file requires a single call to the C API procedure `Imake`. After doing so, space is allocated in the database for an instance of the class specified by the user. It also automatically places the instance in its extent, making it visible for database queries of its class or of any ancestor class. Figure 5.1 shows the Itasca Ada binding implementation of the `Student.Create` function. Attribute initial values are set with the procedures discussed in Section 5.4.2.

This figure illustrates an important point concerning Itasca's ODL preprocessor. As stated in Section 3.5.1, function `Create` in each object's package body must be produced by the preprocessor,

```
function Make(New_Object     : in Uid_Ptr;
              Class_Name     : in String;
              Initial_Values : in String) return Integer is
  Class_Copy   : String(1 .. (Class_Name'Last + 1)) :=
     Class_Name & ASCII.NUL;
  Initial_Copy : String(1 .. (Initial_Values'Last + 1)) :=
     Initial_Values & ASCII.NUL;
begin
  return Imake(New_Object, Class_Copy'Address,
     Initial_Copy'Address);
end Make;

function Create(A_Database    : in Database.Object;
                Last_Name     : in String  := "";
                SSAN          : in Integer := 0;
                Age           : in Integer := 0;
                GPA           : in Float   := 0.0;
                Research_Area : in String  := "")
     return Object is
  New_Object_Ptr : Itasca.Uid_Ptr := Itasca.Allocate_Uid;
  New_Object     : Object;
  Status         : Integer;
begin
  Status := Itasca.Make(New_Object_Ptr, "Student", "");
  if Status /= 0 then
    Itasca.Error_Check(Status);
    return Itasca.No_Object;
  else
    New_Object := New_Object_Ptr.all;
    Set_Value(New_Object, "Last_Name", Last_Name);
    Set_Value(New_Object, "SSAN", SSAN);
    Set_Value(New_Object, "Age", Age);
    Set_Value(New_Object, "GPA", GPA);
    Set_Value(New_Object, "Research_Area", Research_Area);
    return New_Object;
  end if;
end Create;
```

Figure 5.1 Itasca OML Implementation of `Student.Create`

since every ODBMS implements object creation differently. This is a straightforward task for Itasca, as the only required changes to `Create` for a different object package are the string class in the call to `Itasca.Make`, and the individual `Set_Value` calls.

Another important point concerns object identification. In Itasca, an object identifier (OID) is a C character string representing a unique instance, also termed a "unique ID" or Uid. Because type `Object` is a subtype of `System.Address` in the Ada binding approach used in this thesis, and because strings are represented in C by the address of the first character in a sequence terminated by `ASCII.NUL`, it makes perfect sense to represent an Itasca object by the address of the first character in its Uid. In Figure 5.1, the call to `Itasca.Allocate_Uid` dynamically allocates space for an Itasca Uid, which is returned at the end of the function.

Object deletion is straightforward and consists of a single call to `Itasca.Delete_Object`, freeing the Uid string and removing the object from the database.

5.3.2 ObjectStore. Just as ObjectStore's ODL implementation is more complex due to the inability to directly model inheritance, so is the OML implementation of object creation. This process involves first creating the ancestor object or objects and linking each one to its inherited object, then setting the appropriate attribute initial values.

The first step is illustrated in Figure 5.2, the ObjectStore function written by the ODL preprocessor to create a `Student` instance. Creation of the database root is necessary to establish an entry point for its class, and the extent is created in the call to `Student_Set_Pkg.Create`. Memory allocation for an object is performed by calling `Persistent_New` of the appropriate Ada/ObjectStore generic, which is `Student_Pkg` in this case. Note that because these generics must be instantiated at compile time, all lowest-level object creation routines must be produced by the ObjectStore ODL

```
function Create_Student(A_Database : in Database_Object)
  return OS_TYPES.OSTORE_OPAQUE is

  New_Student     : Student.Reference;
  Root            : OSTORE.Database_Root;
  Opaque_Object   : OS_TYPES.OSTORE_OPAQUE;
  Ancestor_Object : OS_TYPES.OSTORE_OPAQUE;

begin
  Root := OSTORE.Database_Root_Find("student_root",
    A_Database);
  if OS_TYPES.Invalid(Root) then
    Root := OSTORE.Database_Create_Root(A_Database,
      "student_root");
    Students := Student_Set_Pkg.Create;
  end if;
  Ancestor_Object := Create_Person(A_Database);
  New_Student     := Student_Pkg.Persistent_New(A_Database);
  Opaque_Object   := Student_Opaque(New_Student);
  Class.Register(Opaque_Object, "Student");
  Student.Set_Value(Opaque_Object, "Ancestor",
    Opaque_Integer(Ancestor_Object));
  Student.Set_Value(Opaque_Object, "Advisor",
    Opaque_Integer(System.NO_ADDR));
  Student_Set_Pkg.Insert_Element(Opaque_Object, Students);
  Student_Pkg.Database_Root_Set_Value(Root, New_Student);
  return Opaque_Object;
end Create_Student;
```

Figure 5.2 Object Creation in Ada/ObjectStore

preprocessor. This constrasts Itasca, whose **Imake** function is placed in non-preprocessed package **Itasca**. After creating the superclass instance, the class of the object is registered in a linked list data structure for later retrieval in other OML operations. Next, the **Ancestor** attribute is updated and the **Advisor** relationship is set to null. The new instance is placed in its extent and the database entry point is updated.

```
function Create(A_Database    : in Database.Object;
                Last_Name     : in String  := "";
                SSAN          : in Integer := 0;
                Age           : in Integer := 0;
                GPA           : in Float   := 0.0;
                Research_Area : in String := "")
     return Object is

  New_Object : Object;

begin
  New_Object := Ada_OS_Query_And_Create.Create_Student(A_Database);
  Set_Value(New_Object, "Last_Name", Last_Name);
  Set_Value(New_Object, "SSAN", SSAN);
  Set_Value(New_Object, "Age", Age);
  Set_Value(New_Object, "GPA", GPA);
  Set_Value(New_Object, "Research_Area", Research_Area);
  return New_Object;
end Create;
```

Figure 5.3 ObjectStore OML Implementation of `Student.Create`

The calling routine of `Create_Student` is shown in Figure 5.3, the `Create` function of package `Student`. As previously stated, both routines must be written by the ObjectStore ODL preprocessor.

The value `New_Object` returned by `Student.Create` is the OID of the new instance. In ObjectStore, this value is the persistent memory address of that object converted from its `Reference` type to `System.Address`. This is an important point which affects the ObjectStore preprocessor's complexity in the implementation of other OML constructs.

5.4 Attributes

5.4.1 ObjectStore. Because an OID in Ada/ObjectStore is that object's persistent memory address, manipulating the attributes of an object first requires translating its address to the

```
function Get_Value(A_Faculty : in Object;
                   Attribute : in String) return Integer is

  Faculty_Object : Reference;

begin
  Faculty_Object := Opaque_Faculty(A_Faculty);
  if Attribute = "SSAN" or
     Attribute = "Age"  or
     Attribute = "Spouse" then
    return Person.Get_Value
      (Integer_Opaque(Faculty_Object.all.Ancestor),
       Attribute);
  elsif Attribute = "Advisees" then
    return Faculty_Object.all.Advisees;
  elsif Attribute = "Ancestor" then
    return Faculty_Object.all.Ancestor;
  else
    Text_IO.Put_Line("Invalid attribute for integer Get_Value");
    return 0;
  end if;
end Get_Value;
```

Figure 5.4 ObjectStore Integer **Get_Value** for Class **Faculty**

appropriate access type. Then, if the attribute is non-inherited, it is referenced with Ada's standard "dot" notation for referencing record fields. If the attribute is inherited, the ancestor pointer is retrieved and translated from **Integer** to **System.Address**, after which control is passed to the corresponding routine in the superclass package.

Figure 5.4 demonstrates getting an integer attribute value in the ObjectStore OML implementation for class **Faculty**. After translating the address type, the appropriate attribute value is returned either directly or by calling the **Get_Value** function in class **Person**. Attribute **Set_Value** operations are implemented in an analogous manner.

```
typedef struct _generic {
     union {
       char        *VAL;
       char        CHAR;
       char        *STRING;
       short       SHORT;
       int         INT;
       long        LONG;
       unsigned    UNSIGNED;
       float       FLOAT;
       double      DOUBLE;
       char        *UID;
     } val;
     struct _generic  *next;
} Igeneric;
```

Figure 5.5 Itasca Attribute Structure Definition

5.4.2 Itasca. Since an Itasca OID is represented as a character string, the memory location of an object's attributes has no direct correlation to its Uid. In fact, an Itasca object attribute is represented as a record with fields defined for every possible system-defined literal. Figure 5.5 shows the Itasca header file definition of structure `Igeneric`, used to model a generic Itasca attribute.

Accessing an attribute value, then, involves invoking Itasca's message passing constructs to access the particular field whose type matches the attribute type for a given Uid. This is performed with the following algorithm:

1. Declare a variable whose data structure matches the attribute structure
2. Set the appropriate type field so that Itasca knows which field to access
3. If performing a `Set`, initialize the field to the desired value
4. Call `Isend`, the function used to send messages to objects in Itasca

```
int get_int_value(uid, attribute)
Iuid uid;
char* attribute;
{
  Icdata val;

  val.type = TYPE_INT;
  if (Isend(&val, attribute, uid, (char *)NULL)) {
    printf("***ITASCA Error: %s\n", ierrstring);
    return 0;
    }
  else
    return val.data.INT;
}


function Get_Value(A_Faculty : in Object;
                   Attribute : in String) return Integer is
begin
  return Itasca.Get_Integer_Attribute(A_Faculty, Attribute);
end Get_Value;
```

Figure 5.6 Itasca Integer **Get_Value** for Class **Faculty**

5. If performing a **Get**, return the value of the appropriate variable field

To avoid defining an Ada type that parallels the Itasca attribute structure, the Ada binding implements all **Set_Value** and **Get_Value** operations in C for each of the attribute types required by the user in an object's package definition. These C routines are accessed using Ada's **pragma** statements in package **Itasca**. Because it is impossible to determine in advance what attributes the user may require, the Itasca auxiliary routines for attribute manipulation are written by the ODL preprocessor and included as necessary. Figure 5.6 shows two of the three calls required in this process. The **Get_Integer_Attribute** function in package **Itasca** simply passes control to its C counterpart **get_int_value**, shown in the figure. Attribute **Set_Value** operations are implemented similarly, using C auxiliary routines and the same three calling levels.

5.5 Relationships

Due to the time constraints of this research, and to facilitate implementing a breadth of Ada binding constructs, only the one-to-one and one-to-many operations are implemented for relationships.

5.5.1 ObjectStore.

Because relationships are stored as integers, their manipulation requires Ada's **`Unchecked_Conversion`** to translate address types to and from integer types. First, however, the **`inverse.dat`** file (discussed in Section 4.4) must be read into a table during relationship initialization. An auxiliary procedure **`Invert`** is then used to extract this information from the table.

Creating a one-to-one relationship—given the source object, related object, and assuming the existence of an inverse—involves the following steps:

1. Determine the class of the source object
2. Determine the relationship's inverse and its multiplicity
3. Set the relationship attribute of the source object to point to the related object
4. If the inverse is one-to-one, update the related object by setting its inverse relationship pointer to the source object
5. If the inverse is one-to-many, update the related object by adding the source object to its inverse relationship set

This process is illustrated in Figure 5.7. If the inverse does not exist, or if the inverse is one-to-one, the procedure reduces to at most two simple **`Set_Value`** operations. Otherwise, if the inverse is one-to-many, the appropriate attribute of type **`Set.Object`** is retrieved and the source object is inserted. Note that because a branch on the source object class is required if the inverse is one-to-many, this routine (and most of the other ObjectStore relationship OML routines) must be produced by the ODL preprocessor.

Creating a one-to-many relationship—given the source object, related set, and assuming the existence of a one-to-one inverse (many-to-many relationships are not implemented)—involves these steps:

1. Determine the class of the related set
2. Determine the relationship's inverse and its multiplicity
3. Set the relationship attribute of the source object to point to the related set
4. Iterate over the related set, updating each element by setting its inverse pointer to the source object

Although straightforward, this procedure is substantially longer than its one-to-one counterpart due to the required branches on the class of the related set. Each of these branches must then iterate over the related set using the appropriate instantiation of `Set_Object`.

The other operations in package **`Relationships`** are also straightforward. Traversals involve simply returning the object or object set pointed to by the given relationship. Deletions involve first traversing the relationship, then deleting the inverse relationship from the related object or set by replacing the pointer with `Opaque_Integer(System.NO_ADDR)`. Note that if the relationship is one-to-many, an iteration over the the related set is required to perform each of the inverse deletions. Adding or removing a one-to-one pair for a one-to-many relationship uses the same

```
procedure Relate_One_To_One_Create
  (Relationship   : in     String;
   An_Object      : in out Persistent_Object;
   Related_Object : in out Persistent_Object) is

  Relationship_Inverse : String_Type := (others => ' ');
  Index                : Integer     := 1;
  One_To_One           : Boolean := False;
  Local_Set            : Set_Object;
  Object_Class         : String(1..100);

begin
  Class.Get_Class(An_Object, Object_Class);
  Invert(Relationship, Relationship_Inverse, Index, One_To_One);
  Set_Value(An_Object, Relationship,
    Opaque_Integer(Related_Object));
  if Relationship_Inverse(1) /= ASCII.NUL then
    if One_To_One then
      Set_Value(Related_Object,
        Relationship_Inverse(1..Index), Opaque_Integer(An_Object));
    else
      Local_Set := Integer_Opaque(Get_Value(Related_Object,
        Relationship_Inverse(1..Index)));
      if Object_Class(1..6) = "Person" then
        Person_Set.Insert_Element(An_Object, Local_Set);
      elsif Object_Class(1..7) = "Student" then
        Student_Set.Insert_Element(An_Object, Local_Set);
      elsif Object_Class(1..7) = "Faculty" then
        Faculty_Set.Insert_Element(An_Object, Local_Set);
      end if;
    end if;
  end if;
end Relate_One_To_One_Create;
```

Figure 5.7 ObjectStore Creation of One-to-one Relationship

algorithm: update the inverse pointer, then add (remove) the related object to (from) the source object's set.

5.5.2 Itasca. Itasca OML relationship implementation is much more flexible than that of ObjectStore, as relationships are defined directly and need not be simulated. Additionally, unlike its ObjectStore counterpart, this package body implementation requires no output from the preprocessor other than the inverse data file.

Before defining the Itasca implementation of package **`Relationships`**, the following auxiliary routines are written to facilitate relationship manipulation:

- **`Update_Uid_Attribute`**
- **`Retrieve_Uid_Attribute`**
- **`Update_Uid_List_Attribute`**
- **`Retrieve_Uid_List_Attribute`**
- **`Add_Uid_To_Set`**
- **`Remove_Uid_From_Set`**

Each of these routines is placed in package **`Itasca`** and is discussed in the following analysis of the Itasca OML relationships implementation.

Creating a one-to-one relationship in Itasca—given the source object, related object, and assuming the existence of an inverse—involves the following steps:

1. Determine the relationship's inverse and its multiplicity
2. Set the relationship attribute of the source object to point to the related object

3. If the inverse is one-to-one, update the related object by setting its inverse relationship pointer to the source object

4. If the inverse is one-to-many, update the related object by adding the source object to its inverse relationship set

Note that these steps are identical to those for creating an ObjectStore one-to-one relationship, except the class of the source object need not be determined. The procedure embodying this process is shown in Figure 5.8. Function `Update_Uid_Attribute` takes as parameters a source Uid, attribute string, and related Uid and makes the appropriate attribute update. Function `Add_Uid_To_Set` adds the related Uid to the set belonging to the source Uid and denoted by the attribute string. Both of these functions return a nonzero value if an error occurs, in which case the `Error_Check` procedure prints the corresponding message.

Creating a one-to-many relationship—given the source object, related set, and assuming the existence of a one-to-one inverse (many-to-many relationships are not implemented)—is also identical to its ObjectStore counterpart, again with the same exception that no object classes need be identified:

1. Determine the relationship's inverse and its multiplicity

2. Set the relationship attribute of the source object to point to the related set

3. Iterate over the related set, updating each element by setting its inverse pointer to the source object

This procedure and the remaining Itasca relationship package body operations are all similar to those reviewed for ObjectStore. They are simpler, however, due to Itasca's abililty to add objects

```
procedure Relate_One_To_One_Create
  (Relationship   : in     String;
   An_Object      : in out Persistent_Object;
   Related_Object : in out Persistent_Object) is

  Relationship_Inverse : String_Type := (others => ' ');
  Index                : Integer     := 1;
  One_To_One           : Boolean     := False;

begin
  Invert(Relationship, Relationship_Inverse, Index, One_To_One);
  Itasca.Error_Check(Itasca.Update_Uid_Attribute(An_Object, Relationship,
    Related_Object));

  if Relationship_Inverse(1) /= ASCII.NUL then
    if One_To_One then
      Itasca.Error_Check(Itasca.Update_Uid_Attribute(Related_Object,
        Relationship_Inverse(1..Index), An_Object));
    else
      Itasca.Error_Check(Itasca.Add_Uid_To_Set(Related_Object,
        Relationship_Inverse(1..Index), An_Object));
    end if;
  end if;
end Relate_One_To_One_Create;
```

Figure 5.8 Itasca Creation of One-to-one Relationship

```
DEFADAREC(Persistent_Object)
  DEFIELD(Object_ID,INTEGER)
ENDADAREC(Persistent_Object)
```

Figure 5.9 ObjectStore Class for Storing OIDs

to sets without determining their classes at compile time. Functions **Retrieve_Uid_Attribute** and **Retrieve_Uid_List_Attribute** are used to traverse one-to-one and one-to-many relationships respectively, while **Update_Uid_List_Attribute** initializes an attribute of type **set-of** to the **Uid_list** parameter.

5.6 Persistent Names

The unique Ada OML implementations of operations **Name_Object** and **Lookup_Object** in package **Database** are discussed in the following sections.

5.6.1 ObjectStore. The ObjectStore analogy to persistent naming is creating a database root and setting (or getting) its value. This can be done for any object class declared persistent in the schema file. One solution for assigning a persistent name, then, is to determine the class of the object, create a database root, and set its value to the input string. A persistent name lookup would then require finding the object reference associated with the given name, get its value using the appropriate generic instantiation based on its class, and convert it to the required **System.Address** type for the calling routine.

A more elegant solution is possible, however. All ObjectStore OIDs are simply memory addresses, using the same number of bits as an integer type (see Section 4.4.2.2). Therefore, a new object class can be introduced to the schema file with a single attribute of type **Integer**, which will actually store OIDs. Figure 5.9 shows the required declarations in the schema file for this class.

```
procedure Name_Object(An_Object  : in OS_TYPES.OSTORE_OPAQUE;
                      Name       : in String;
                      A_Database : in Database_Object) is

  New_Persistent_Object : Persistent_Object_Ptr;
  Root                  : OSTORE.Database_Root;

begin
  Root := OSTORE.Database_Root_Find(Name,
    A_Database);
  if OS_TYPES.Invalid(Root) then
    Root := OSTORE.Database_Create_Root(A_Database,
      Name);
    New_Persistent_Object :=
      Persistent_Object_Pkg.Persistent_New(A_Database);
    New_Persistent_Object.all.Object_ID := Opaque_Integer(An_Object);
    Persistent_Object_Pkg.Database_Root_Set_Value(Root,
      Opaque_Persistent(An_Object));
  else
    Text_IO.Put("Error: Persistent name '");
    Text_IO.Put(Name);
    Text_IO.Put_Line("' already exists in the database.");
  end if;
end Name_Object;
```

Figure 5.10 ObjectStore OML Implementation of `Name_Object`

The `Name_Object` procedure is then a simplified version of the steps outlined in the alternate solution. An additional instantiation of `OSTORE_GENERICS` is required to access the database root operations. The final version of this procedure is shown in Figure 5.10. Function `Lookup_Object` is similar, returning the output from `Database_Root_Get_Value`.

5.6.2 Itasca. Itasca does not have an explicit facility for assigning persistent names to objects, so the simplest solution is to define a new ancestor class with an attribute storing a set of persistent name strings. This attribute is then inherited by every subclass, giving each object

```
attr1 = IUmake_attribute("Persistent_Names", DOMAIN, "(set-of string)",
  END_ARGS);

if (Imake_class(&uid, "Persistent_Object", ATTRIBUTES, attr1, END_ARGS))
  printf("  Schema Error: %s\n", ierrstring);
IUfree_generic(attr1, TYPE_VAL_LIST);
```

Figure 5.11 Itasca Class for Storing Persistent Names

instance a potential set of persistent names. Figure 5.11 illustrates the C API code creating the new class `Persistent_Object`, from which classes `Person`, `Student`, and `Faculty` are derived.

The `Name_Object` procedure adds the new persistent name to an object's `Persistent_Names` attribute, provided the new name is unique. This is done by first performing a lookup on the new name. If no valid object is returned, a call is made to auxiliary routine `add_string_to_set`, which adds an input string to an attribute of type `(set-of string)` for a given object.

Function `Lookup_Object` is slightly more complex. Unlike ObjectStore, where a persistent name lookup is implemented as true non-associative retrieval, the Itasca implementation requires a query based on the given name. The C API `Iselect_any_star` function returns the first object satisfying a given predicate, which must be entered as a Lisp expression in string form (Itasca queries are discussed in more detail in Chapter VI).

Figure 5.12 shows the Itasca implementation of `Lookup_Object`. Parameter `Name` is placed in a Lisp query string which uses the `some` keyword to check if the given name is in an object's persistent name list. Memory for a new object is then allocated, and the query function is called. If a valid object was found it is returned; otherwise, the null object is returned. In this case an error message is not printed, since the `Lookup_Object` function is also called by `Name_Object`.

```
function Lookup_Object(Name       : in String;
                       A_Database : in Object := Current)
  return Persistent_Object is

  New_Object_Ptr : Itasca.Uid_Ptr;
  String_Break   : Integer := Name'last + 34;
  Expression     : String(1..String_Break);

begin
  Expression(1..String_Break) := "(equal (some Persistent_Names) " &
    '"' & Name(1..Name'last) & '"' & ")";
  New_Object_Ptr := Itasca.Allocate_Uid;
  Itasca.Error_Check(Itasca.Select_Any_Star(New_Object_Ptr,
    "Persistent_Object", Expression));
  return New_Object_Ptr.all;
end Lookup_Object;
```

Figure 5.12 Itasca Implementation of Persistent Name Retrieval

5.7 Collections

Class **Array_Type** of the Ada binding is not implemented in this thesis. Of the remaining collection subclasses, only a subset of the operations are implemented, and these are shown in Figure 5.13. The following sections discuss the ODBMS implementations of these constructs.

5.7.1 ObjectStore. The original Ada/ObjectStore interface package did not provide access to ObjectStore's collection facility; however, Li Chou provided this access in his extensions to Ada/ObjectStore (4:3-11–3-13). He implemented the collection package as a generic, requiring the Ada type to be stored, a pointer to this type, and the collection typespec (defined in header file `osada.hh`) to be passed in as instantiation parameters. In our example, the first two instantiation parameters correspond to **X.Attributes** and **X.Reference** for object class **X**. The individual collection subclasses are not immediately instantiable in Ada/ObjectStore; they are, however, in-

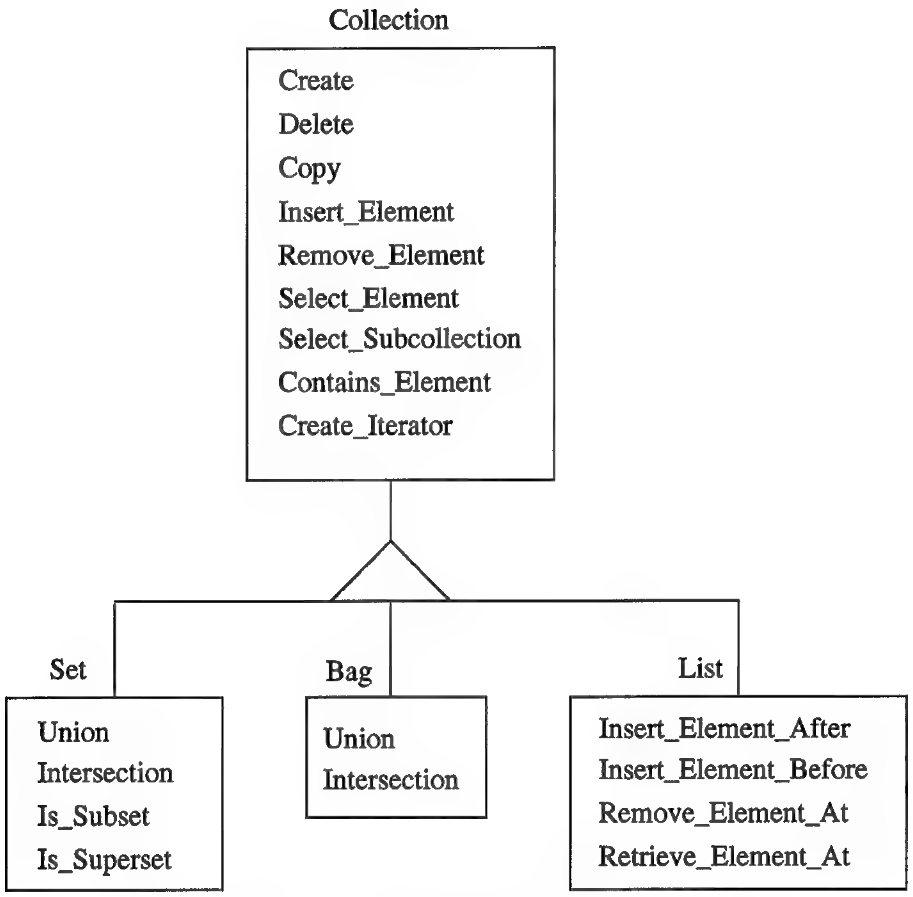

Figure 5.13 Implemented Collection Operations

stantiated by creating a new collection with the appropriate keywords, discussed in more detail in the following sections.

The ObjectStore implementation of collections requires output from the ODL preprocessor. Ada/ObjectStore uses the string class instantiation parameter to determine the appropriate generic for a given operation. This appears in the familiar `if-then-else` format seen in earlier sections of this chapter.

The implemented operations for the subclasses are now discussed. Operations of class collection (repeated in each of the three subclasses) are reviewed first, followed by operations unique to each subclass.

5.7.1.1 Collection.

- `Create`. This function determines the behavior of the collection, and consequently it identifies the particular collection subclass. Because each function is different, it is left for discussion in the subclasses.

- `Delete`. Transfers control to the Ada/ObjectStore procedure `OS_COLLECTION_DELETE`, then removes the stored address from the registered list by calling `Class.Deregister`.

- `Copy`. Transfers control to the Ada/ObjectStore procedure `OS_COLLECTION_COPY`.

- `Insert_Element`. Uses `Unchecked_Conversion` to convert the address type to its corresponding access type, then transfers control to procedure `OS_COLLECTION_INSERT`.

- `Remove_Element`. Converts the object, then transfers control to the Ada/ObjectStore procedure `OS_COLLECTION_REMOVE`.

- `Select_Element`. Parses the query string to translate it from Ada to C syntax, then returns the output from function `OS_COLLECTION_QUERY_PICK`.

- `Select_Subcollection`. Parses the query string, then returns the output from function `OS_COLLECTION_QUERY`.

- `Contains_Element`. Returns the output from function `OS_COLLECTION_CONTAINS` using the converted object.

- `Create_Iterator`. Returns the output from function `OS_CURSOR_CREATE` for the appropriate cursor package.

5.7.1.2 Set.

- `Create`. Initializes a locally created persistent set to the output from `OS_COLLECTION_CREATE`, called with the initialization string `"maintain_cursors"`. The class of the objects contained in the set is then stored by calling `Class.Register`, and the local set is returned.

- `Union`. Copies one of the sets to a locally created persistent set, calls `OS_COLLECTION_UNION` using the local set and the second set, then returns the local set.

- `Intersection`. Same as `Union`, calling `OS_COLLECTION_INTERSECT` as the Ada/ObjectStore operation.

- `Is_Subset`. Determines whether the first set is a subset of the second set by returning the output from the boolean function `OS_COLLECTION_LESS_THAN_OR_EQUAL`.

- `Is_Superset`. Same as `Is_Subset`, but returns the output from the Ada/ObjectStore boolean function `OS_COLLECTION_GREATER_THAN_OR_EQUAL`.

5.7.1.3 Bag.

- `Create`. Initializes a locally created persistent bag to the output from `OS_COLLECTION_CREATE`, called with the initialization string `"maintain_cursors | allow_duplicates"`. The bag class is then stored by calling `Class.Register`, and the local bag is returned.

- `Union`. Passes control to `OS_COLLECTION_UNION`. Because duplicates are allowed, the cardinality of the resulting bag equals the sum of the cardinalities of the original bags.

- `Intersection`. Passes control to `OS_COLLECTION_INTERSECT`. An intersection operation over two bags is described in ODMG-93 as creating a new empty bag, iterating over the first bag, and at each iteration, if the iterating object is an element of the second bag, that object is placed in the result (1:34). Note that this description implies the operation is not commutative; that is, if `A = {a, a, b}` and `B = {a, b}`, `A` intersect `B` equals set `A`. `B` intersect `A`, however, equals set `B`. Due to the nature of Ada/ObjectStore's intersection operation, though, the number of occurrences of every element in the resulting bag equals the minimum of the number of occurrences for that element in either of the two operands. The Ada/ObjectStore implementation, then, is always commutative, and will return a copy of set `B` in this example.

5.7.1.4 List.

- **`Create`**. Initializes a locally created persistent list to the output from `OS_COLLECTION_CREATE`, called with the initialization string "`maintain_cursors | allow_duplicates | maintain_order`". The list class is then stored by calling `Class.Register`, and the local list is returned.

- `Insert_Element_After`. This procedure is shown in Figure 5.14 for class `Person`. A comparable procedure does not exist in Ada_ObjectStore, so a local list is first created, along with a local iterator. Each element of the input list is placed in the local list by calling `OS_COLLECTION_INSERT_LAST`, until the desired position is reached. The input object is then inserted, and the remaining elements of the input list are inserted, preserving their original order. Finally, the input list is assigned to the local list.

- `Insert_Element_Before`. Essentially the same as the previous procedure, the only exception being one less object inserted in the local list before the input object is inserted.

- `Remove_Element_At`. Iterates over the input list until the desired location is reached, then calls `OS_COLLECTION_REMOVE` using the converted object.

- `Retrieve_Element_At`. Iterates over the input list until the desired location is reached, then returns the object at that location.

5.7.2 Itasca. Unlike ObjectStore, Itasca does not natively support collections. As a result, implementing collections for Itasca required more effort than simply passing control to an appropriate existing operation.

The Itasca construct with the closest resemblence to a collection is the `set-of` specifier, for attributes that can be assigned multiple values. This construct is used to store one-to-many relationship pointers and persistent names, discussed in earlier sections. To manipulate collections and store them persistently in Itasca, a new object class `Collection` is required, whose only attribute `Value` is a set of type `Persistent_Object`. The C API code creating this class is shown in Figure 5.15.

```
procedure Insert_Element_After(An_Object : in     Persistent_Object;
                               Position  : in     Integer;
                               A_List    : in out Object) is

    Local_List     : Object            := Create;
    Local_Iterator : Iterator_Object   := Create_Iterator(A_List);
    Local_Object   : Persistent_Object;
    Index          : Integer           := 1;
    Cardinality    : Integer;

  begin
    if Class_Name = "Person" then
      Cardinality :=
        Local_Integer(Person_List.OS_COLLECTION_CARDINALITY(A_List));
      Person_Iterator.First(Local_Iterator, Local_Object);
      while Index <= Position and
            Person_Iterator.More(Local_Iterator) loop
        Index := Index + 1;
        Person_List.OS_COLLECTION_INSERT_LAST(Local_List,
          Opaque_Person(Local_Object));
        Person_Iterator.Next(Local_Iterator, Local_Object);
      end loop;
      Person_List.OS_COLLECTION_INSERT_LAST(Local_List,
        Opaque_Person(An_Object));
      while Index <= Cardinality and
            Person_Iterator.More(Local_Iterator) loop
        Index := Index + 1;
        Person_List.OS_COLLECTION_INSERT_LAST(Local_List,
          Opaque_Person(Local_Object));
        Person_Iterator.Next(Local_Iterator, Local_Object);
      end loop;
      A_List := Local_List;
    elsif Class_Name = "Student" then
      .
      .
      .
  end Insert_Element_After;
```

Figure 5.14 ObjectStore Implementation of Insert_Element_After

```
attr1 = IUmake_attribute("Value", DOMAIN, "(set-of Persistent_Object)",
    END_ARGS);

  if (Imake_class(&uid, "Collection", SUPERCLASSES, "Persistent_Object",
      ATTRIBUTES, attr1, END_ARGS))
    printf("  Schema Error: %s\n", ierrstring);
  IUfree_generic(attr1, TYPE_VAL_LIST);
```

Figure 5.15 Itasca Class for Storing Collections

```
Iuid_list all_uids, ulp;

Iselect_star(&all_uids, "Student",
  QUERY_EXPRESSION, "(> GPA 3.0)", END_ARGS);
for (ulp = all_uids;  ulp;  ulp = ulp->next)
  printf("UID is %s\n", ulp->val.UID);
```

Figure 5.16 Code Fragment Using Type `Iuid_list`

The Itasca **set-of** specifier actually creates an attribute of C API type **Iuid_list**, the data structure also used to store query results and one-to-many relationship pointers. This type is simply a pointer to type **Igeneric** (Figure 5.5). An example of a C API code fragment using type **Iuid_list** can be seen in Figure 5.16. The variable **all_uids** points to a linked list containing the results of the query selecting all students with a GPA greater than 3.0. Another **Iuid_list** variable **ulp** iterates over the linked list and is used in printing each object's identifier.

Collections, then, are implemented using this same linked list data structure stored as an attribute to an instance of class `Collection`. It is important to note that because it is a linked list, the **set-of** attribute specifier does not in itself represent a "true" set, as duplicates are allowed. Therefore, for collection subclass **Set**, a new element is inserted only if it is not already a member of the associated `Uid_List` attribute.

The biggest challenges to implementing collections in this manner lie in the `Select_Element` and `Select_Subcollection` operations. Because Itasca does not have a facility for testing whether a given predicate is true or false for a particular object, a new object method definition is required. Several factors make this a difficult task:

1. Itasca methods must be implemented in Lisp
2. The parsed predicate string such as `(> Age 30)` must be modified so that the object's actual `Age` value is referenced
3. The modified predicate string must then be evaluated

The first factor requires a working knowledge of the Lisp programming language, and the third factor involves simply passing the modified predicate to Lisp's `eval` function, returning the result. The second factor, however, requires writing a Lisp function to transform the predicate string into a valid Lisp predicate returning true or false.

To accomplish this, method `Predicate` is defined for class `Person`, thus inherited for each of the `Student` and `Faculty` subclasses. Itasca C API methods allow the user to define a variable referencing the object for which the method is to be executed (7:115). By convention, this variable is typically referred to as `self`. Additionally, the value of an object's attribute is obtained by calling the Itasca `send` function. Therefore, the sample predicate `(> Age 30)` must be transformed to `(> (send Age self) 30)` before it can be successfully evaluated.

Figure 5.17 shows the C code defining method `Predicate`. Parameter `predicate-string` represents the inital predicate string. The Lisp form `labels` allows the local recursive function `modify-predicate` to be defined, making the necessary transformation by replacing each attribute

```
Idef_method(&string_rtn, "Predicate", "Self", "Person", FALSE,
  "(predicate-string)",
  "(labels ((modify-predicate (predicate) \
             (cond \
              ((null predicate) nil) \
              ((atom predicate) predicate) \
              ((listp (first predicate)) \
               (cons (modify-predicate (first predicate)) \
               (modify-predicate (rest predicate)) )) \
              ((member (first predicate) (attributes (class-of self))) \
               (cons `(send ',(first predicate) ,self) \
               (modify-predicate (rest predicate)) )) \
              (t (cons (first predicate)  \
                 (modify-predicate (rest predicate)) )) ))) \
        (if (eval (modify-predicate predicate-string)) \
             1 0))" );
```

Figure 5.17 Definition of Method `Predicate`

with its `(send <attribute> self)` counterpart. The modified predicate is then passed to `eval`, and if it evaluates true, an integer 1 is returned; otherwise, the function returns an integer 0.

Finally, auxiliary routine `evaluate_predicate` is needed to call the method and process the results. This routine is shown in Figure 5.18. A quote is added to the front of the string to delay evaluation of the predicate until it is passed to `eval`. If the method is invoked without error, the integer result is returned.

The remaining implementation details for Itasca collections are discussed in the following subsections.

5.7.2.1 Collection.

- `Create`. Calls `Itasca.Make` to create and return a new instance of class `Collection`.
- `Delete`. Calls `Itasca.Delete_Object` to purge the object from memory.
- `Copy`. Calls `Itasca.Copy_Object` to create and return a copy of the given collection object.

```
int evaluate_predicate(uid, predicate)
Iuid uid;
char* predicate;
{
  Icdata val;
  Istring new_predicate;

  new_predicate = (char*)IUmalloc(strlen(predicate)+2);
  strcpy(new_predicate, "'");
  strcat(new_predicate, predicate);
  val.type = TYPE_INT;
  if (Isend(&val, "Predicate", uid, new_predicate)) {
    printf("  Isend Error: %s\n", ierrstring);
    return 0;
  }
  else
    return val.data.INT;
}
```

Figure 5.18 Auxiliary Routine For Evaluating Object Predicates

- `Insert_Element`. This operation depends on the collection subclass, and is discussed individually in the sections that follow.

- `Remove_Element`. First tests if the input object exists in the `Iuid_list`, and if so passes control to `Itasca.Remove_Uid_From_Set` to locate and remove it from the `Value` attribute of the collection object.

- `Select_Element`. Converts all single quotes to double quotes in the predicate string. Next, the input collection is iterated over until the predicate evaluates true on one of the objects. If the predicate cannot be satisfied, or if the collection is empy, a null object is returned.

- `Select_Subcollection`. First, calls `Create` to persistently allocate a new collection object. After converting single quotes to double quotes, iterates over the input collection, adding each object satisfying the predicate to the local collection, which is then returned.

- `Contains_Element`. Iterates over the collection, testing if each object is equal to the input object by calling C API routine `Ieqo`. When either a match is found or the collection is exhausted, true or false is returned, respectively.

- `Create_Iterator`. Returns the `Iuid_list` data structure referenced by attribute `Value` of the given collection instance.

5.7.2.2 Set.

- **Insert_Element**. If the input object is not already a member of the set, calls **Itasca.Add_Uid_To_Set** to add the object to the **Value** attribute of the collection object.

- **Union**. Creates a local iterator for the first set and copies the second set to an initially empty persistently allocated result set. Iterates over the first set and inserts each element into the result, which is then returned.

- **Intersection**. Creates a local iterator for the first set and creates an initially empty persistently allocated result set. Iterates over the first set and if the iterating object is also contained in the second set, that object is inserted into the result. The result set is then returned.

- **Is_Subset**. Creates a local iterator for the first set and initializes the local boolean **Subset** to true. Iterates over the first set, and if any object is not also contained in the second set, **Subset** is assigned the value of false. The value of **Subset** is returned.

- **Is_Superset**. Returns the value of **Is_Subset**, called with the parameters reversed.

5.7.2.3 Bag.

- **Insert_Element**. Calls **Itasca.Add_Uid_To_Set** regardless of whether the object already exists in the bag.

- **Union**. Same as for class **Set**. Note that if an object appears in both bags, it will appear at least twice in the resulting bag.

- **Intersection**. Same as for class **Set**. Duplicates are retained in the resulting bag. Unlike the Ada/ObjectStore version, this operation is not commutative, as it is implemented exactly as stated in ODMG-93.

5.7.2.4 List.

- **Insert_Element**. Same as for class **Bag**. Because of the behavior of the C API routine **Itasca.Add_Uid_To_Set**, the new element is always inserted at the front of the input list.

- **Insert_Element_After**. Calls auxiliary routine **insert_element_after_position**, which initializes a new **Iuid_list** node and links it into the list after the indicated position.

- **Insert_Element_Before**. Calls auxiliary routine **insert_element_before_position**, which initializes a new **Iuid_list** node and links it into the list before the indicated position.

- **Remove_Element_At**. Calls auxiliary routine **remove_object_by_position**, which moves to the indicated position in the **Iuid_list** and removes the object at that location.

- **Retrieve_Element_At**. Creates an iterator for the list and moves to the object at the input position. This object is returned, unless the position is invalid, in which case a null object is returned.

5.8 Iterators

Iterator operations **First**, **Last**, **Next**, and **More** are implemented for both ODBMSs in this thesis.

5.8.1 ObjectStore.

Similar to collections, the ObjectStore implementation of each operation in package **Iterator** reduces to a simple passing of control to Ada/ObjectStore's cursor facility. Cursors are implemented as generics, and an instantiation for each class in the database is included at the beginning of the package. Control is passed in the familiar **if-then-else** construct based on the string instantiation parameter of package **Iterator**.

5.8.2 Itasca.

Itasca's implementation requires writing auxiliary routines dereferencing the appropriate pointers in the linked list structure of a collection. The **First** operation, then, ensures the pointer is valid and returns the corresponding Uid with routine **Itasca.Uid_Field**; **Last** traverses the list and returns the last valid Uid. Operation **Next** invokes auxiliary routine **Itasca.Next_Field**, and **More** simply ensures a valid pointer with **Itasca.Valid_Pointer**.

5.9 Transactions

All transaction operations presented in the Ada binding are implemented with the exception of **Checkpoint**, as an analogous operation does not exist in either ODBMS. The four remaining operations are all supported by the transaction facilities of Ada/ObjectStore, including transaction nesting. As a result, these routines correspond directly to their Ada/ObjectStore counterparts.

Itasca does not explicitly support nested transactions, but as identified by Moyers, it does provide a way to simulate them using sessions (14:4-7). The **`Start`** operation, then, corresponds to C API function **`Iopen_session`**. To keep track of the current nesting level, the global integer variable **`Transaction_Count`** is incremented upon transaction commencement and decremented upon transaction completion. Whether a transaction aborts or commits, it is concluded with the **`Iclose_session`** routine. If the transaction commits, it is preceded by the **`Icommit`** routine, and if it aborts, by the **`Iabort`** routine. **`Abort_To_Top_Level`** is accomplished by repeated calls to **`Iabort`** and **`Iclose_session`**, decrementing the counter each time until it reaches zero.

5.10 Database Operations

This section discusses the database operations of the Ada binding with the exception of **`Name_Object`** and **`Lookup_Object`** (discussed in Section 5.6) and **`Query`** (Chapter VI).

5.10.1 Initialization and Finalization.

Both ODBMS implementations require initialization processing. The ObjectStore implementation of **`Initialize_Interface`** consists of a call to **`Class.Initialize`**, clearing the data structure used to determine object classes based on OID, and a call to Ada/ObjectStore procedure **`INIT_ADA_INTERFACE`**. For Itasca, the initialize procedure consists of a single call to **`Itasca.Connect`**, which opens a connection to the Itasca image.

No finalization is required in ObjectStore, and its version of **`Stop_Interface`** is a no-op. Itasca requires a call to **`Itasca.Disconnect`**, closing the connection.

5.10.2 File Manipulation. Opening an ObjectStore database file requires calling Ada/ ObjectStore's `DATABASE_OPEN` function, which returns a pointer to the file. This pointer is stored in package variable `Current` and returned by the `Database.Open` function.

Because only one database file may be open in Itasca at any given time, `Current` has no meaning in its implementation. More importantly, however, an Ada database application implemented with Itasca and operating on two database files simultaneously will almost certainly provide erroneous results. This limits the number of truly portable applications that can be written for these two Ada binding implementations. However, it is an inevitable limitation and demonstrates that the portability of any ODMG-93 language binding is ultimately dependent upon the underlying ODBMS implementations.

As expected, a database file is closed in ObjectStore by calling `OSTORE.DATABASE_CLOSE`, passing it the file pointer stored in `Current` or passed in by the user. Itasca has no construct for closing a database file. If a new file is to be opened it is simply restarted, making it the new current database file. Therefore, its version of operation `Close` in package `Database` is a no-op.

5.11 Transient Object Manipulation

ObjectStore treats the manipulation of transient and persistent data in exactly the same way. Therefore, a distinction is not required when manipulating transient and persistent instances of objects in the ObjectStore OML implementation. As stated in Section 3.4.2, however, the user is restricted to manipulating only noninherited attributes of transient instances.

Because object identifiers and attributes are represented internally to Itasca, transient and persistent object manipulation is not identical. If the user wishes to manipulate transient object instances in a database application, the Itasca implementation must be modified to let the user manipulate transient attributes using Ada's standard "dot" notation rather than Itasca's message passing constructs. Possible modifications to incorporate this ability are explored in Chapter VII.

5.12 OML Implementation Results

Consistent with the ObjectStore implementation of Ada ODL, its implementation of object creation must incorporate the inheritance simulation techniques discussed in Section 5.3. Consequently, an ODL preprocessor more complex than the one for Itasca is required. Furthermore, attribute manipulation requires resolving these simulation techniques, and since attributes are also manipulated in relationship operations, the ObjectStore ODL preprocessor is involved in each of these constructs. Because objects and attributes are manipulated in Itasca using Lisp's message passing utilities, its preprocessor remains relatively uninvolved in the Itasca OML implementation.

A similar situation exists for collections. In Ada/ObjectStore, collections are implemented as generics with instantiation parameters representing the attribute record of the object to be collected and a pointer to this record. Only one instantiation parameter is required in the Ada binding: a string representing the class to be collected. As a result, collections must be instantiated in the ObjectStore OML implementation at compile time, requiring output from the preprocessor. Itasca treats collections as object instances, so again its preprocessor remains uninvolved.

The persistent name implementations are straightforward for both ODBMSs. The ability to handle persistent names as database roots makes ObjectStore's version more efficient, while the Itasca version requires the extra overhead of a database query for both the lookup and name operations. Each ODBMS implementation allows more than one persistent name to be defined for a particular object, and each ensures that a persistent name assigned in the database is unique.

An important note regarding the Itasca manipulation of attributes involves version 2.2 used for this thesis. In this version, the type of the `Icdata` variable in Figure 5.6 must be determined before calling `Isend`. However, in Itasca version 2.3 the system automatically determines the return type based on the attribute parameter of `Isend`. Incorporating this capability by upgrading to a more recent version will most likely decrease the number of auxiliary routines used in manipulating attributes, and the requirement of preprocessor output for Itasca attribute manipulation may even be removed altogether.

5.13 Summary

With the exception of the collection subclass packages, the implementation of each Ada OML package body for Itasca is less complex than its ObjectStore counterpart, which requires substantial output from the preprocessor. This results in a cleaner, more flexible, and more complete OML implementation for Itasca. ObjectStore's database root operations, however, provide for a more efficient implementation of persistent naming than that of Itasca, and transient object manipulation requires no modifications to the ObjectStore implementation. The next chapter reviews the final test of the Ada binding: Object Query Language.

VI. OQL Implementation

6.1 Overview

This chapter discusses Ada Object Query Language (OQL) implementations for Itasca and ObjectStore, comparing and summarizing the results of each one.

6.2 Introduction

As mentioned in Chapter III, ODMG-93 OQL and consequently Ada OQL are both powerful facilities, allowing the user to retrieve elements and subsets of extents, collections of literals, and even dynamic structures created "on the fly." Due to the time constraints and experimental nature of this research, the only operations implemented here include selection of a subset of an extent, and selection of a singleton object using the **`element`** keyword. Furthermore, both versions allow only predicates involving immediate attribute values as opposed to attributes of objects participating in relationships.

6.3 ObjectStore

Ada/ObjectStore has available two functions for querying a collection: `OS_COLLECTION_QUERY` and `OS_COLLECTION_QUERY_PICK`. The parameters for each include the collection to be queried, the pointer type (in string form) against which to apply the selection string, the query selection string itself, and the database file. To illustrate, Figure 6.1 shows the Ada/ObjectStore version of the two queries in Figure 3.10, based on our current object model implementation.

Consequently, executing an OQL query in Ada/ObjectStore requires the following algorithm:

```
Over21 : Person_Set.Object :=
  Person_Set.OS_COLLECTION_QUERY(Persons,
    "Person*", "Age > 21", Database.Current);

Jones : Student.Object :=
  Student_Set.OS_COLLECTION_QUERY_PICK(Students,
    "Student*", "!strcmp(Ancestor->Last_Name, 'Jones')",
    Database.Current);
```

Figure 6.1 Example Ada/ObjectStore Queries

1. Extract the extent and predicate from the query string, and determine if the query will return an object or a collection
2. Based on the extent, determine the object pointer type for applying the predicate string
3. Translate the predicate from Ada OQL syntax to the required C format, incorporating the chosen technique for simulating inheritance
4. Call the `QUERY` or `QUERY_PICK` function, as appropriate, returning its result

The first step is performed using a simple parsing procedure located in package `Ostore_Strings`; its specification is shown in Figure 6.2. The boolean value of parameter `Element_Query` is determined by the presence of keyword `element` at the beginning of the expression. The value of `Extent` is determined next by scanning to the `in` keyword if it exists; if it does not, the query must be a simple one-word extent query as in example query six of Figure 3.8. The predicate is determined by returning the characters following keyword `where`.

Step two in the transformation algorithm incorporates procedure `Extent_To_Class`, which does a simple lookup of the class corresponding to the given extent. Step three is also implemented here using a simple lookup, but in a non-simulated environment this will be the most complex step to implement. Not only must the Ada operators such as "=" and "/=" be interchanged with their C

```
procedure Parse_Query(Expression       : in     String;
                      Predicate        : in out Ostore_String_Type;
                      Predicate_Index  : in out Integer;
                      Extent           : in out Ostore_String_Type;
                      Extent_Index     : in out Integer;
                      Element_Query    :    out Boolean);
```

Figure 6.2 Procedure Specification for Ada/ObjectStore Query String Parser

counterparts ("==" and "!="), string equality must also be tested using a call to **strcmp**, as shown in Figure 6.1. Further complicating matters is the requirement that the inheritance simulating technique be resolved to reference any inherited attributes. Finally, step four is a simple branch based on the value of **Element_Query** returned from the query string parser.

The final query function is shown in Figure 6.3. Procedures **Query** and **Query_Pick** in preprocessed package **Ada_OS_Query_And_Create** parse the predicate string and call their Ada/ObjectStore counterparts in the appropriate generic instantiation representing the extent.

6.4 Itasca

The Itasca implementation is almost identical to Ada/ObjectStore, with some important simplifications. Here, the query functions are **Iselect_star** and **Iselect_any_star** of the C API. Parameters to each include a **Uid** or **Uid_List** pointer to store the result, the class (in string form) to be queried, and the predicate string. Figure 6.4 shows the calls to Ada package **Itasca**, linking to the corresponding Itasca C API **Select** functions, for the two queries of Figure 3.10.

Consequently, executing an OQL query in Itasca requires the following algorithm:

1. Extract the extent and predicate from the query string, and determine if the query will return an object or a collection

```
function Query(Expression : in String;
               A_Database : in Database.Object := Current)
  return System.Address is

  Predicate       : Ostore_Strings.Ostore_String_Type := (others => ' ');
  Predicate_Index : Integer := 0;
  Extent          : Ostore_Strings.Ostore_String_Type := (others => ' ');
  Extent_Index    : Integer := 0;
  Element_Query   : Boolean := false;

begin
  Ostore_Strings.Parse_Query(Expression, Predicate, Predicate_Index,
    Extent, Extent_Index, Element_Query);
  if Element_Query then
    return Ada_OS_Query_And_Create.Query_Pick(A_Database,
      Ostore_Strings.Extent_To_Class(Extent(1..Extent_Index)),
      Predicate(1..Predicate_Index));
  else
    return Ada_OS_Query_And_Create.Query(A_Database,
      Ostore_Strings.Extent_To_Class(Extent(1..Extent_Index)),
      Predicate(1..Predicate_Index));
  end if;
end Query;
```

Figure 6.3 ObjectStore Implementation of Ada OQL Query Function

```
  New_Set_Ptr : Itasca.Uid_List_Ptr := Itasca.Allocate_Uid_List;
  Itasca.Error_Check(Itasca.Select_Star(New_Set_Ptr, "Person",
    "(> Age 21)"));

  New_Object_Ptr : Itasca.Uid_Ptr := Itasca.Allocate_Uid;
  Itasca.Error_Check(Itasca.Select_Any_Star(New_Object_Ptr, "Student",
    "(equalp Last_Name 'Jones')"));
```

Figure 6.4 Example Itasca Queries

2. Translate the predicate from Ada OQL syntax to the required Lisp format

3. Allocate dynamic memory for a `Uid` or create a new collection object to store the result, as appropriate

4. Call the `Select_Star` or `Select_Any_Star` function, and return the dereferenced `Uid` or the updated collection object

Step one is performed using a string parser identical to the ObjectStore string parser, except located in package `Itasca_Strings`. Step two is also simulated with a simple lookup, although in a non-simulated environment, an infix to prefix parser is required, incorporating the Lisp function calls representing their Ada counterparts. If it is an element query, step three is performed with an auxiliary routine calling `malloc` to return the necessary memory; otherwise `Create` is used to persistently allocate space for a new collection object. The appropriate Itasca selection routine is called and the result is either dereferenced (element query) or stored in attribute `Value` of the collection object (collection query). Figure 6.5 illustrates this process.

6.5 OQL Implementation Results

Once again, the Itasca implementation of an Ada binding construct is simpler than its corresponding ObjectStore implementation. The ObjectStore version requires output from the preprocessor due to the generic nature of the collection facility. These generics must be instantiated at compile time before the calls to `QUERY` and `QUERY_PICK` can be made.

The need to simulate inheritance in Ada/ObjectStore further complicates its implementation. While both versions require a parser to translate query expressions from Ada to the syntax of the underlying ODBMS, Ada/ObjectStore must also resolve the referencing of inherited attributes

```
function Query(Expression : in String;
               A_Database : in Database.Object := Current)
  return System.Address is

  Predicate       : Itasca_Strings.Itasca_String_Type := (others => ' ');
  Predicate_Index : Integer := 0;
  Extent          : Itasca_Strings.Itasca_String_Type := (others => ' ');
  Extent_Index    : Integer := 0;
  Element_Query   : Boolean := false;

  New_Set_Ptr     : Itasca.Uid_List_Ptr;
  New_Object_Ptr  : Itasca.Uid_Ptr;
  Result_Set      : Local_Set.Object;
  Quote_Converted : Itasca_Strings.Itasca_String_Type := (others => ' ');
  Converted_Index : Integer := 0;

begin
  Itasca_Strings.Parse_Query(Expression, Predicate, Predicate_Index,
    Extent, Extent_Index, Element_Query);
  if Predicate_Index > 0 then
    Itasca_Strings.Convert_Quotes(
      Itasca_Strings.Parse_Expression(Predicate(1..Predicate_Index)),
        Quote_Converted, Converted_Index);
  end if;
  if Element_Query then
    New_Object_Ptr := Itasca.Allocate_Uid;
    Itasca.Error_Check(Itasca.Select_Any_Star(New_Object_Ptr,
      Itasca_Strings.Extent_To_Class(Extent(1..Extent_Index)),
      Quote_Converted(1..Converted_Index)));
    return New_Object_Ptr.all;
  else
    Result_Set := Local_Set.Create;
    New_Set_Ptr := Itasca.Allocate_Uid_List;
    Itasca.Error_Check(Itasca.Select_Star(New_Set_Ptr,
      Itasca_Strings.Extent_To_Class(Extent(1..Extent_Index)),
      Quote_Converted(1..Converted_Index)));
    Itasca.Error_Check(Itasca.Update_Uid_List_Attribute(Result_Set, "Value",
      New_Set_Ptr.all));
    return Result_Set;
  end if;
end Query;
```

Figure 6.5 Itasca Implementation of Ada OQL Query Function

using the chosen simulation technique. Because Itasca has an explicit construct for specifying inheritance, its implementation of the query facility is understandably simpler.

In this research, both versions use a string parser to extract the element, extent, and predicate information from the original query string. This is due to the inability of Itasca and ObjectStore to process queries in the **`select-from-where`** form. By 1995, however, vendors representing over 80 percent of the ODBMS market will have implemented this capability (2:3). At that time, the only required modification to the query string will be its translation of the predicate string from Ada syntax to the syntax required by the underlying system, and in the case of ObjectStore, resolution of the inheritance simulation technique.

6.6 Summary

Queries over extents, returning either collections or singleton elements, were implemented for both ObjectStore and Itasca, and the Itasca version's relative simplicity is attributed primarily to its construct for specifying inheritance in a language that does not natively support it. The following chapter discusses overall results for this research project as well as recommendations for future research.

VII. Conclusions and Recommendations

7.1 Overview

This chapter summarizes the thesis and discusses several conclusions that can be drawn from examining it as a whole, in particular the degrees to which the binding and implementation goals are satisfied. The results of a simple performance comparison are summarized and explained, and recommendations are made for future research.

7.2 Summary of Research

This thesis produced an Ada language binding to the ODMG-93 object database standard, implementing a subset of these operations for the Itasca and ObjectStore object database management systems (ODBMSs). The driving justification is the need for Ada programmers to be able to access the functionality of object databases, the latest generation of DBMSs. The complete binding was proposed in Chapter III, and Chapter IV identified the optimal implementation approach as well as the implementation of Ada Object Definition Language (ODL). Chapters V and VI presented the binding implementations for Ada Object Manipulation Language (OML) and Ada Object Query Language (OQL), respectively. Tables 7.1 through 7.5 list the constructs presented in ODMG-93 along with their implementation status in this research.

Table 7.1 Implementation of ODMG-93 Object Management and Manipulation Operations

		Tested with Itasca/ObjectStore	
Operation Class	**ODMG-93 Construct**	**Yes**	**No**
Database	open	√	
	close	√	
	contains_object?		√
	lookup_object	√	
Transaction	begin	√	
	commit	√	
	abort	√	
	checkpoint		√
	abort_to_top_level	√	
Object	create_instance[1]	√	
	delete	√	
	create_extent	√	
	same_as?		√
	name_object	√	
Attributes	set_value	√	
	get_value	√	
Methods	invoke		√
	return		√
	return_abnormally		√

[1]Clustering not implemented

Table 7.2 Implementation of ODMG-93 Relationship Operations

		Tested with Itasca/ObjectStore	
Operation Class	**ODMG-93 Construct**	**Yes**	**No**
One-to-one	create	✓	
	delete	✓	
	traverse	✓	
One-to-Many	create	✓	
	delete	✓	
	add_one_to_one	✓	
	remove_one_to_one	✓	
	traverse	✓	
Many-to-many	create		✓
	delete		✓
	traverse		✓

Table 7.3 Implementation of ODMG-93 Iteration Operations

		Tested with Itasca/ObjectStore	
Operation Class	**ODMG-93 Construct**	**Yes**	**No**
Iterator	next	✓	
	first	✓	
	last	✓	
	more?	✓	
	reset		✓
	delete		✓

Table 7.4 Implementation of ODMG-93 Collection Operations

		Tested with Itasca/ObjectStore	
Operation Class	**ODMG-93 Construct**	**Yes**	**No**
Collection	create	√	
	delete	√	
	copy	√	
	insert_element	√	
	remove_element	√	
	remove_element_at		√
	replace_element_at		√
	retrieve_element_at		√
	select_element	√	
	select	√	
	exists?		√
	contains_element?	√	
	create_iterator	√	
Set	union	√	
	intersection	√	
	difference		√
	is_subset?	√	
	is_proper_subset?		√
	is_superset?	√	
	is_proper_superset?		√
Bag	union	√	
	intersection	√	
	difference		√
List	insert_element_after	√	
	insert_element_before	√	
	insert_first_element		√
	insert_last_element		√
	remove_element_at	√	
	remove_first_element		√
	remove_last_element		√
	replace_element_at		√
	retrieve_element_at	√	
	retrieve_first_element		√
	retrieve_last_element		√
Array	insert_element_at		√
	remove_element_at		√
	replace_element_at		√
	retrieve_element_at		√
	resize		√

Table 7.5 Implementation of ODMG-93 Query Operations

		Tested with Itasca/ObjectStore	
Operation Class	**ODMG-93 Construct**	**Yes**	**No**
Query	select[2]	✓	
	select distinct[3]	✓	
	select struct		✓
	select distinct struct		✓
	select couple		✓
	define		✓
	element[4]	✓	
	mod		✓
	abs		✓
	first		✓
	last		✓
	listtoset		✓
	element		✓
	flatten		✓
	intersect		✓
	union		✓
	except		✓
	for all		✓
	exists		✓
	sort		✓
	count		✓
	sum		✓
	min		✓
	max		✓
	avg		✓
	group		✓

[2]Only implemented for subsets selected over an extent
[3]See previous footnote
[4]Only implemented for type `Object` selected over an extent

7.3 Conclusions

Numerous conclusions can be made regarding the designing and testing of an Ada language binding to ODMG-93. Each one is identified and discussed below.

7.3.1 Binding Goals. As explained in Chapter I, the three Ada binding goals sought in this thesis were completeness, transparency, and portability. In the following sections, each goal is reviewed along with an estimate of the degree to which it was satisfied.

7.3.1.1 Completeness. Completeness means the application using the Ada binding can access all of the object database functionality specified in ODMG-93. The Ada binding incorporates each construct presented in the object database standard; therefore, it is considered 100% complete.

7.3.1.2 Transparency. Like the other language bindings presented in ODMG-93, the Ada binding assumes the existence of an ODL preprocessor specific to an underlying ODBMS implementation. This preprocessor has the ability to scan an Ada application and the corresponding object class definitions, creating database schemas and auxiliary source code. Because the user is shielded from the details of the ODBMS implementation language, transparency is 100% satisfied in the Ada binding.

7.3.1.3 Portability. Portability is also 100% satisfied, as the binding is presented independent of an underlying system. Furthermore, a single test driver was used to test each ODBMS implementation. Of course, the primary goal of this thesis was a portable object database interface for Ada, and every effort was directed toward ensuring portability was achieved.

7.3.2 Implementation Goals. As explained in Chapter I, two goals were sought for an ODBMS implementation of the Ada binding: expandability and maintainability. The design method proposed and selected in Chapter IV, the indirect method, was chosen based on its inherent support of both goals. Because a modular approach was taken, requiring the Ada binding to invoke an Ada package which in turn linked to the C routines implementing each ODBMS construct, the implementation developer can easily add new utilities as they become available from the vendor. Additionally, if an error is detected in a vendor routine, only one modification should be required due to the fact that each vendor routine is linked in only one location. Both of these goals are considered 100% satisfied.

7.3.3 Comparison of Preprocessor Dependence. As previously stated, a preprocessor was assumed to exist for both ODBMSs for the purpose of creating the database schema and producing auxiliary source code. However, without exception, the ObjectStore version required a preprocessor to create every package body implementing an Ada binding construct in this thesis. The only additional preprocessor output in the Itasca implementation involved the manipulation of attributes, and with the new 2.3 version release, even this requirement can possibly be removed.

Several reasons account for ObjectStore's increased preprocessor requirements, not the least of which is the strong typing of its ODBMS implementation language. The C language implementation accessing ObjectStore's functionality in this thesis necessitates the creation of parallel data types before attribute values may be manipulated. Since attribute manipulation occurs in every Ada binding construct, it logically follows that the ObjectStore implementation is highly dependent on a preprocessor. Itasca, on the other hand, has no requirement for declaring parallel data types, primarily due to the fact that Itasca's ODBMS implementation language is Lisp, a weakly typed language. Any ODBMS implementation of the Ada binding interface presented in this thesis will require extensive preprocessor output if the ODBMS implementation language is strongly typed. Consequently, an automated version of the Ada binding incorporating functional ODL preprocessors will be more difficult to implement for a strongly typed ODBMS than for one that is weakly typed, such as Itasca.

Another reason a preprocessor is so important for the ObjectStore implementation involves its method of object identification and how that method relates to attribute manipulation. The Object Identifier (OID) of an ObjectStore object is that object's location in persistent memory, while the OID of an Itasca object is a unique character string that has no direct correlation to its memory address.[5] Consequently, when an Ada/ObjectStore object attribute is manipulated, the OID must first be converted to an access type referencing the Ada record that parallels the corresponding C pointer type. The attribute is then accessed using Ada's dot notation for accessing record fields. Itasca, on the other hand, manipulates all its attribute values using the C API `Isend` command, passing it the character string OID. This operation requires no preprocessor output whatsoever.

Finally, inheritance must be simulated in Ada/ObjectStore, and when attribute manipulation is required, the method of simulation must be resolved. An example can be seen in the `Get_Value` and `Set_Value` operations for any of the three object classes implemented in this thesis. The preprocessor was required to produce a branch of an `if-then-else` statement to detect an inherited attribute and make an additional call to the superclass package using the address of the ancestor object. In Itasca, inherited attributes are manipulated in exactly the same way as non-inherited attributes, again requiring no ouput from the preprocessor.

[5]Of course, the advantage is faster performance for ObjectStore, discussed in Section 7.3.4

Table 7.6 Performance Results for ODBMS Implementations

Resource (seconds)	**Ada Interface to Itasca**	**Average**	**Ada Interface to ObjectStore**	**Average**	**Percent Change**
CPU time	2.440 2.360 2.740	2.510	1.770 1.540 1.320	1.540	-39.0
Elapsed time	44.328 42.201 44.345	43.620	10.858 10.571 10.552	10.660	-76.0

These three reasons help explain why ObjectStore is a difficult ODBMS to access using languages other than C or C++. Itasca has none of the above requirements, and consequently it is ideal for accessing in a variety of languages. This is Itasca's major strength, and it's why Itasca Systems, Inc. has produced application program interfaces (APIs) for C, C++, Lisp, and CLOS. A commercially produced Ada API is also currently under development (5:1–4).

7.3.4 Performance Comparison. Package **STATIS_ADA**, the statistics monitoring package used by Li Chou (4:4-1–4-5), is included in the test driver to measure the performance of the two ODBMS implementations. Each driver was executed three times on comparable platforms during low usage periods. Table 7.6 illustrates the amount of CPU time and total elapsed time for each run along with the averages of these values. Please note that this is by no means a controlled scientific analysis of performance; it is simply a measure of the time requirements for the two implementations of the test driver.

Table 7.6 shows that the ObjectStore implementation required approximately 39% less CPU time than the Itasca version and executed an average of 76% faster for the user. These results are entirely expected, and can be accounted for primarily by the way ObjectStore manipulates persistent data. With its Virtual Memory Mapping Architecture, ObjectStore can handle persistent data as fast as transient data by using memory mapping, caching, and clustering techniques to optimize data access (16:5–6). The results shown here are confirmed by Halloran, who compared the ObjectStore, Matisse, and Itasca ODBMSs and determined ObjectStore was indeed the fastest of the three (6:88–107).

An additional reason for Itasca's slower performance is the nature of object identification. As previously stated, the OID of an ObjectStore object is that object's address in persistent memory, while the OID of an Itasca object is a unique character string that has no direct correlation to its memory address. Therefore, Itasca requires an additional table lookup for the address of an object once its OID is determined; ObjectStore does not. Once again, performance was not a major issue in this thesis, but if it becomes one in a future ODBMS implementation, these factors should be carefully considered.

7.3.5 Transient Object Manipulation. As stated in Chapter V, transient object instances cannot be manipulated in the Itasca implementation. ObjectStore, however, treats the manipulation of transient and persistent data in exactly the same way, and therefore a distinction is not required when manipulating either of the two. Along with its improved performance, this is one of the most important advantages of the ObjectStore implementation. Any ODBMS implementation incorporating parallel data types for each object class will manipulate transient objects more easily than an implementation that performs attribute manipulation internally to the ODBMS, although preprocessor dependence will necessarily be increased.

While the user is allowed to create both transient and persistent objects in the Ada binding, only non-inherited attributes of transient instances may be manipulated due to Ada 83's inability to model inheritance. This restriction applies to any ODMG-93 language binding produced for a non-object-oriented language. If inheritance is modeled or simulated in the ODBMS implementation, transient objects will not be able to take advantage of this capability.

7.3.6 Ada 9X. Ada 9X, the latest version of Ada 83 currently in the final standardization process, allows a programmer to specify inheritance between packages (3:385–406). Although it may appear at first glance that designing and implementing the binding in Ada 9X would remove the need to simulate inheritance in the ObjectStore implementation, this is not the case. To be sure, Ada 9X provides a more standardized mechanism for communicating object inheritance to the ODL preprocessor than the method proposed by Moyers (14:B-3–B-4). However, inheritance is specified in an ObjectStore application by defining C++ classes using C++ inheritance syntax, and this implementation uses ObjectStore's C interface library to access the ODBMS functionality. Until a version of the Ada 9X compiler is released that can link with C++ modules, the preprocessor will still need to simulate inheritance even if the Ada binding is written in Ada 9X.

7.4 Recommendations for Future Research

This topic may be further explored and refined in several ways, each of which is discussed in the following sections.

7.4.1 Binding Ada 9X to ODMG-93. While Ada 9X is unable to automatically model inheritance in an ODBMS implementation language without such a capability, Ada 9X does have the potential for two dramatic improvements to the Ada binding. An immediate improvement is that transient instances would be able to manipulate all attributes, whether inherited or non-inherited, as inheritance would be communicated not only to the ODL preprocessor but also to the Ada compiler.

Most importantly, however, an Ada 9X binding would allow ODMG-93 constructs to incorporate the majority of error-checking at compile-time, rather than at runtime. In fact, an approach quite similar to the C++ binding's Ref-based approach (1:84–85) could be achieved by declaring a root object package **`Persistent`** from which all persistence-capable database classes could be derived (11:3–4, 12). This feature and the ability to model inheritance in transient objects would result in a more complete Ada binding and should be further examined.

7.4.2 Removing the Need to Simulate Inheritance. Two possible alternatives exist for removing the need to simulate inheritance in the ObjectStore implementation of the current Ada binding. The most feasible option lies in the definition of object classes using `os_ada.hh`, discussed in Chapter IV. This header file uses a simple set of macros to create a schema file defining the C parallel data types for an Ada record. The resulting schema file is actually C++ source code, as required by the ObjectStore Schema Generator (15:92). Although it was never explored in this thesis, an additional set of macros could possibly be introduced to the header file, explicitly specifying inheritance to the C++ compiler and eliminating the need for inheritance simulation. Whether or not ObjectStore's C interface library will then be able to manipulate these attributes is unclear, but it is an area that deserves consideration.

A related alternative is to use the C++ "mangled names" referred to by Li Chou (4:3-6–3-7). Chou noted that C++ is designed as a preprocessor for C, converting C++ source code to C before compilation. As a result, all C++ routines can be represented by their mangled names, or the names given to the C versions of the C++ routines. While it would involve a great deal of research, ObjectStore's C++ library could possibly be accessed in this manner. This alternative coupled with the new macro definition discussed above may be a viable option to the inheritance simulation technique used in this thesis.

7.4.3 Improving Testing Completeness. Due to the time constraints of this research, numerous operations in the Ada binding were not tested with ObjectStore and Itasca. To further test the Ada binding, the following unimplemented constructs should be added.

7.4.3.1 Object Creation. Specific details regarding object creation are not addressed in the ODMG-93 discussion of its Object Definition Language (ODL), but an example is provided in its discussion of the C++ binding (1:93). An important feature mentioned is that an existing OID may be passed to the `Create_Object` routine as a clustering object, specifying that the new object should be placed near the clustering object. The C++ binding states the exact interpretation of "near" is implementation-defined. To improve performance for the user, clustering should be incorporated in the Ada binding, as it is supported by both ObjectStore (16:63–64) and Itasca (7:113).

7.4.3.2 Object Manipulation. Probably the most dramatic way to improve the thoroughness of each test implementation is to implement many-to-many relationships. Because the Ada binding defines these relationships in terms of their component one-to-many relationships, several of the existing relationship operations can probably be reused.

Constructs which will likely be more difficult to implement are methods, or operations defined for instances of a particular object class. In our data model, one such method might be `Birthday`, defined on class `Person`, to increment the age attribute. Although methods were required for the Itasca implementation of collections, the concept of an instance method to modify the state of an object was not addressed in the implementations presented in this thesis. One option for implementing methods in Itasca is to first translate the Ada method source code into Lisp, then define a method for the corresponding Itasca class using the resulting Lisp code. For ObjectStore the user-defined source code can simply be executed for the given object, as attribute manipulation is performed in Ada and not in the ODBMS implementation language. A major advantage here is that the Ada code will not have to be translated to the ODBMS implementation language. Both of these techniques should be explored in future research.

7.4.3.3 Collections. Unimplemented operations for collection subclass `Set` include `Difference`, `Is_Proper_Subset`, and `Is_Proper_Superset` (1:32–33). The `Difference` operation is also omitted from subclass `Bag` (1:33–34). These implementations will most likely be similar to the existing set theoretical operations, involving iterations over one of the operands while examining containment of the iterating object in the other operand.

Additionally, `List` operations not implemented include `Insert_First_Element`, `Insert_Last_Element`, `Remove_First_Element`, `Remove_Last_Element`, `Retrieve_First_Element`, `Retrieve_Last_Element`, and `Replace_Element_At` (1:34–35). These implementations will involve producing auxiliary routines for the Itasca version, and preprocessed routines for Ada/ObjectStore, consistent with the existing implemented `List` operations.

Finally, collection subclass **`Array_Type`** is completely unimplemented. Adding this package will require extensive research, as ObjectStore does not support arrays in the C interface, and Itasca does not support collections at all.

7.4.3.4 Queries. Both Ada OQL implementations currently allow the selection of a subset of an extent and selection of a singleton, but valid predicates may only involve immediate attribute values as opposed to attributes of objects participating in relationships. For example, in either implementation there is no way to query the database for all faculty members advising a student over a given age. This restriction should be relaxed by implementing a robust query facility which would be of more practical use. Because of the required OQL format, an SQL-like parser will almost certainly need to be incorporated. Fully functional expression parsers should also be used to transform selection criteria from postfix to infix for Itasca, and to replace the Ada binary operators with their Lisp and C counterparts for Itasca and ObjectStore, respectively. As illustrated in Table 7.5, the query facility represents the area of the Ada binding implementations with the most room for improved testing completeness and should be given top priority in future analysis.

7.4.3.5 Error Checking and Exception Handling. Because this was the first design and implementation of a complete object database interface for Ada, most of the ODMG-93 constructs were included in the Ada binding assuming error-free input parameters. In the real world, error checking is an important part of any major application and should be included here.

An exception model is incorporated in the standard as the primary method of communicating errors to the user (1:26). Ada's robust exception facility should be mapped to the ODMG-93 model as the primary error-checking mechanism.

7.4.4 Incorporating Ada Binding Annexes. Several features unique to the underlying ODBMSs were not included in the Ada binding due to their omission from the object database standard. One example is Itasca's dynamic schema evolution (DSE) capability, allowing attribute definitions for existing classes to be modified during execution, and also allowing new object classes to be inserted at any location in the inheritance hierarchy (7:59). ODMG-93 discusses DSE only as a future revision to the current standard (1:44). However, to take advantage of this valuable facility in Itasca, an annex to the Ada binding could be implemented.

An additional feature supported by both ODBMSs but also addressed as a future revision to the standard is object versioning, a configuration management facility allowing early states of an object to be recorded while also allowing alternative states to be simultaneously available. Both Itasca (7:81–87) and ObjectStore (16:227–265) support versioning, but the current standard does not (1:44). Annexes could also be written to access object versioning in both of these ODBMSs.

7.4.5 Performance Benchmarking. A major justification for designing a language binding to ODMG-93 is that a corresponding application can execute with the implementation providing the best performance, in terms of either time or memory requirements. Table 7.6 showed the time requirements of executing both test driver implementations. Because this was an informal test, as performance was not a major issue in this thesis, more thorough tests should be performed in a controlled environment for a wide variety of applications. Additionally, a technique for measuring the memory requirements of the test driver should be developed. Such tests would provide valuable performance information regarding these ODBMS implementations.

7.4.6 Revisions to the Standard. The Ada binding designed and implemented in this thesis incorporates the newly proposed object database standard ODMG-93. As the standard evolves, changes to the binding may be required. The standard should be closely monitored and any necessary updates should be added and tested.

7.4.7 Revisions to the ODBMSs. Like any major commercial software product, ODBMSs are constantly evolving, and the two used in this thesis are no exceptions. New and more powerful version releases should be incorporated in this research project to improve the performance and the elegance of the current test implementations. An excellent starting point would be to incorporate the aforementioned Ada API under development by EVB. Although Ada/ObjectStore is no longer supported by Object Design, Inc. (9:1), updates to the ObjectStore ODBMS itself may prove to be more efficient and more easily interfaced with Ada than Ada/ObjectStore. Technical summaries for future versions of Itasca and ObjectStore should be analyzed for facilities that can possibly improve the implementations developed here.

7.5 Final Comments

This thesis proves that a portable, transparent and complete object database interface for Ada is indeed achievable. With the Ada language binding to ODMG-93, Ada programmers can access Itasca and ObjectStore without the need to understand the programming languages implementing these systems. Additional implementations are certainly achievable as well, and Ada programmers are well on their way to writing portable applications to take advantage of the power and reusability of the latest generation of database management systems.

Appendix A. Ada Binding to ODMG-93

This appendix contains the complete Ada binding for the proposed ODMG-93 standard, organized by Ada package.

A.1 Database

```
with System;

package Database is

  subtype  Object            is System.Address;
  subtype  Persistent_Object is System.Address;

  Current : Object;

  procedure Initialize_Interface;
  procedure Stop_Interface;

  function Open(Database_Name : in String) return Object;

  procedure Close(A_Database : in Object := Current);

  function Contains(A_Database : in Object := Current;
                    An_Object  : in Persistent_Object)
    return Boolean;

  procedure Name_Object(An_Object  : in Persistent_Object;
                        Name       : in String;
                        A_Database : in Object := Current);

  function Lookup_Object(Name : in String;
                         A_Database : in Object := Current)
    return Persistent_Object;

  function Query(Expression : in String;
                 A_Database : in Database.Object := Current)
    return System.Address;

end Database;
```

A.2 Transaction

```
with System;
```

```
package Transaction is

  subtype Object is System.Address;

  function Start return Object;

  procedure Commit_Txn(A_Transaction : in Object);

  procedure Abort_Txn(A_Transaction : in Object);

  procedure Checkpoint(A_Transaction : in Object);

  procedure Abort_To_Top_Level;

end Transaction;
```

A.3 Set

```
with System, Database;

generic
  Class_Name : String;
package Set is

  subtype Object            is System.Address;
  subtype Persistent_Object is System.Address;
  subtype Iterator_Object   is System.Address;

-----------------------------------------------------------------
-- Operations inherited from Collection:

  function Create(A_Database : in Database.Object := Database.Current)
    return Object;

  procedure Delete(A_Set : in out Object);

  function Copy(A_Set : in Object) return Object;

  procedure Insert_Element(An_Object : in     Persistent_Object;
                           A_Set     : in out Object);

  procedure Remove_Element(An_Object : in     Persistent_Object;
                           A_Set     : in out Object);

  procedure Remove_Element_At(Position : in     Integer;
                             A_Set    : in out Object);

  procedure Replace_Element_At(An_Object : in     Persistent_Object;
                               Position  : in     Integer;
                               A_Set     : in out Object);

  function Retrieve_Element_At(Position  : in Integer;
                               A_Set     : in Object)
    return Persistent_Object;

  function Select_Element(A_Set     : in Object;
                          Predicate : in String)
    return Persistent_Object;

  function Select_Subcollection(A_Set     : in Object;
                                Predicate : in String)
    return Object;

  function Exists(A_Set     : in Object;
                  Predicate : in String)
    return Boolean;

  function Contains_Element(A_Set     : in Object;
                            An_Object : in Persistent_Object)
    return Boolean;

  function Create_Iterator(A_Set : in Object) return Iterator_Object;

-----------------------------------------------------------------
-- Operations specific to Set:

  function Union(Set_One : in Object;
                 Set_Two : in Object) return Object;

  function Intersection(Set_One : in Object;
                        Set_Two : in Object) return Object;

  function Difference(Set_One : in Object;
                      Set_Two : in Object) return Object;

  function Is_Subset(Set_One : in Object;
                     Set_Two : in Object) return Boolean;

  function Is_Proper_Subset(Set_One : in Object;
                            Set_Two : in Object) return Boolean;
  function Is_Superset(Set_One : in Object;
                       Set_Two : in Object) return Boolean;

  function Is_Proper_Superset(Set_One : in Object;
                              Set_Two : in Object) return Boolean;

end Set;
```

A.4 Bag

```
with System, Database;

generic
  Class_Name : String;
package Bag is

  subtype Object            is System.Address;
  subtype Persistent_Object is System.Address;
  subtype Iterator_Object   is System.Address;

-----------------------------------------------------------------
-- Operations inherited from Collection:

  function Create(A_Database : in Database.Object := Database.Current)
    return Object;

  procedure Delete(A_Bag : in out Object);

  function Copy(A_Bag : in Object) return Object;

  procedure Insert_Element(An_Object : in     Persistent_Object;
                           A_Bag     : in out Object);

  procedure Remove_Element(An_Object : in     Persistent_Object;
                           A_Bag     : in out Object);

  procedure Remove_Element_At(Position : in     Integer;
                              A_Bag    : in out Object);

  procedure Replace_Element_At(An_Object : in     Persistent_Object;
                               Position  : in     Integer;
                               A_Bag     : in out Object);

  function Retrieve_Element_At(Position  : in Integer;
                           A_Bag     : in Object)
    return Persistent_Object;

  function Select_Element(A_Bag     : in Object;
                          Predicate : in String)
    return Persistent_Object;

  function Select_Subcollection(A_Bag     : in Object;
                                Predicate : in String)
    return Object;

  function Exists(A_Bag     : in Object;
                  Predicate : in String)
    return Boolean;

  function Contains_Element(A_Bag     : in Object;
                            An_Object : in Persistent_Object)
    return Boolean;

  function Create_Iterator(A_Bag : in Object) return Iterator_Object;

-----------------------------------------------------------------
-- Operations specific to Bag:

  function Union(Bag_One : in Object;
                 Bag_Two : in Object) return Object;

  function Intersection(Bag_One : in Object;
                        Bag_Two : in Object) return Object;

  function Difference(Bag_One : in Object;
                      Bag_Two : in Object) return Object;

end Bag;
```

A.5 List

```
with System, Database;

generic
  Class_Name : String;
package List is

  subtype Object            is System.Address;
  subtype Persistent_Object is System.Address;
  subtype Iterator_Object   is System.Address;

-----------------------------------------------------------------
-- Operations inherited from Collection:

  function Create(A_Database : in Database.Object := Database.Current)
    return Object;

  procedure Delete(A_List : in out Object);

  function Copy(A_List : in Object) return Object;

  procedure Insert_Element(An_Object : in     Persistent_Object;
                           A_List    : in out Object);

  procedure Remove_Element_At(Position : in     Integer;
                              A_List   : in out Object);

  procedure Replace_Element_At(An_Object : in     Persistent_Object;
                               Position  : in     Integer;
                               A_List    : in out Object);

  function Retrieve_Element_At(Position  : in Integer;
                               A_List    : in Object)
    return Persistent_Object;

  procedure Remove_Element(An_Object : in     Persistent_Object;
                           A_List    : in out Object);

  function Select_Element(A_List    : in Object;
                          Predicate : in String)
    return Persistent_Object;

  function Select_Subcollection(A_List    : in Object;
                                Predicate : in String)
    return Object;

  function Exists(A_List    : in Object;
                  Predicate : in String)
    return Boolean;

  function Contains_Element(A_List    : in Object;
                            An_Object : in Persistent_Object)
    return Boolean;
  function Create_Iterator(A_List : in Object) return Iterator_Object;

----------------------------------------------------------------
-- Operations specific to List:

  procedure Insert_Element_After(An_Object : in     Persistent_Object;
                                 Position  : in     Integer;
                                 A_List    : in out Object);

  procedure Insert_Element_Before(An_Object : in     Persistent_Object;
                                  Position  : in     Integer;
                                  A_List    : in out Object);

  procedure Insert_First_Element(An_Object : in     Persistent_Object;
                                 A_List    : in out Object);

  procedure Insert_Last_Element(An_Object : in     Persistent_Object;
                                A_List    : in out Object);

  procedure Remove_First_Element(A_List : in out Object);

  procedure Remove_Last_Element(A_List : in out Object);

  function  Retrieve_First_Element(A_List : in Object)
    return Persistent_Object;

  function  Retrieve_Last_Element(A_List : in Object)
    return Persistent_Object;

end List;
```

A.6 Array_Type

```
with System, Database;

generic
  Class_Name : String;
package Array_Type is

  subtype Object            is System.Address;
  subtype Persistent_Object is System.Address;
  subtype Iterator_Object   is System.Address;

----------------------------------------------------------------
-- Operations inherited from Collection:
  function Create(A_Database : in Database.Object := Database.Current)
    return Object;

  procedure Delete(An_Array : in out Object);

  function Copy(An_Array : in Object) return Object;

  procedure Insert_Element(An_Object : in     Persistent_Object;
                           An_Array  : in out Object);

  procedure Remove_Element(An_Object : in     Persistent_Object;
                           An_Array  : in out Object);

  procedure Remove_Element_At(Position : in     Integer;
                              An_Array : in out Object);

  procedure Replace_Element_At(An_Object : in     Persistent_Object;
                               Position  : in     Integer;
                               An_Array  : in out Object);

  function Retrieve_Element_At(Position  : in Integer;
                               An_Array  : in Object)
    return Persistent_Object;

  function Select_Element(An_Array  : in Object;
                          Predicate : in String)
    return Persistent_Object;

  function Select_Subcollection(An_Array  : in Object;
                                Predicate : in String)
    return Object;

  function Exists(An_Array  : in Object;
                  Predicate : in String)
    return Boolean;

  function Contains_Element(An_Array  : in Object;
                            An_Object : in Persistent_Object)
    return Boolean;

  function Create_Iterator(An_Array : in Object) return Iterator_Object;

-----------------------------------------------------------------
-- Operations specific to Array:
  procedure Insert_Element_At(An_Object : in     Persistent_Object;
                              Position  : in     Integer;
                              An_Array  : in out Object);

  procedure Resize(New_Size : in     Integer;
                   An_Array : in out Object);

end Array_Type;
```

A.7 Iterator

```
with System;

generic
  Class : String;
package Iterator is

  subtype Object            is System.Address;
  subtype Persistent_Object is System.Address;

-----------------------------------------------------------------
-- Manipulate Iterators.  Note that Iterators cannot be created
-- in this package, as the Create_Iterator operation is defined
-- in the collection to be iterated over.

  function More(An_Iterator : in Object) return Boolean;

  procedure First(An_Iterator : in out Object;
                  An_Object   :    out Persistent_Object);

  procedure Last(An_Iterator : in out Object;
                 An_Object   :    out Persistent_Object);

  procedure Next(An_Iterator : in out Object;
                 An_Object   :    out Persistent_Object);

  procedure Reset(An_Iterator : in out Object);

  procedure Delete(An_Iterator : in out Object);

end Iterator;
```

A.8 Relationships

```
with System;

package Relationships is

  subtype Persistent_Object is System.Address;
  subtype Set_Object        is System.Address;

  procedure Initialize;

-----------------------------------------------------------------

  procedure Relate_One_To_One_Create
    (Relationship   : in     String;
     An_Object      : in out Persistent_Object;
     Related_Object : in out Persistent_Object);

  procedure Relate_One_To_One_Delete
    (Relationship : in     String;
     An_Object    : in out Persistent_Object);

  function Relate_One_To_One_Traverse
    (Relationship : in String;
     An_Object    : in Persistent_Object)
       return Persistent_Object;

-----------------------------------------------------------------

  procedure Relate_One_To_Many_Create
    (Relationship : in     String;
     An_Object    : in out Persistent_Object;
     Related_Set  : in     Set_Object);

  procedure Relate_One_To_Many_Delete
    (Relationship : in     String;
     An_Object    : in out Persistent_Object);

  procedure Relate_One_To_Many_Add_One_To_One
    (Relationship  : in     String;
     An_Object     : in out Persistent_Object;
     Object_To_Add : in out Persistent_Object);

  procedure Relate_One_To_Many_Remove_One_To_One
    (Relationship     : in     String;
     An_Object        : in out Persistent_Object;
     Object_To_Remove : in out Persistent_Object);
  function Relate_One_To_Many_Traverse
    (Relationship : in String;
     An_Object    : in Persistent_Object)
       return Set_Object;

------------------------------------------------------------------

  procedure Relate_Many_To_Many_Delete
    (Relationship : in     String;
     An_Object    : in out Persistent_Object);

  procedure Relate_Many_To_Many_Add_One_To_One
    (Relationship  : in     String;
     An_Object     : in out Persistent_Object;
     Object_To_Add : in out Persistent_Object);

  procedure Relate_Many_To_Many_Remove_One_To_One
    (Relationship     : in     String;
     An_Object        : in out Persistent_Object;
     Object_To_Remove : in out Persistent_Object);

  procedure Relate_Many_To_Many_Add_One_To_Many
    (Relationship : in     String;
     An_Object    : in out Persistent_Object;
     Set_To_Add   : in     Set_Object);

  procedure Relate_Many_To_Many_Remove_One_To_Many
    (Relationship  : in     String;
     An_Object     : in out Persistent_Object;
     Set_To_Remove : in     Persistent_Object);

  procedure Relate_Many_To_Many_Remove_All_From
    (Relationship : in     String;
     An_Object    : in out Persistent_Object);

end Relationships;
```

Bibliography

1. Atwood, Tom and others. *The Object Database Standard: ODMG-93*. San Mateo, CA: Morgan Kaufmann Publishers, 1994.
2. Atwood, Tom and others. "Repsonse to the ODMG-93 Commentary Written by Dr. Won Kim of UniSQL, Inc.," *SIGMOD RECORD*, *23*(3):3–7 (September 1994).
3. Cernosek, Gary. "ROMAN-9X: A Technique For Representing Object Models in Ada 9X Notation," *Association for Computer Machinery*, 385–406 (1993).
4. Chou, Li. *Object-Oriented Database Access from Ada*. MS thesis, AFIT/GCS/ENG/93M-01, Air Force Institute of Technology (AETC), March 1993.
5. EVB Software Engineering, Inc., 5303 Spectrum Drive, Frederick, MD 21701. *Itasca/Ada Application Programming Interface for Ada*, 1994.
6. Halloran, Timothy J. *Performance Measurement of Three Commercial Object-oriented Database Management Systems*. MS thesis, AFIT/GCS/ENG/93D-12, Air Force Institute of Technology (AETC), December 1993.
7. Itasca Systems, Inc., Minneapolis, MN. *C API User Manual for Release 2.2*, 1993.
8. Kim, Won. "Observations on the ODMG-93 Proposal for an Object-Oriented Database Language," *SIGMOD RECORD*, *23*(1) (March 1994).
9. Kincaid, Tom. "Ada Integration with ObjectStore." Electronic Mail #SE007496_O, March 1994.
10. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts*. New York: McGraw-Hill, Inc., 1991.
11. Lindsay, Stephen R. and Mark A. Roth. *An Ada Binding for ODMG-93*. Technical Report AFIT/EN/TR/94-09, Wright-Patterson AFB, OH 45433-7765: Air Force Institute of Technology (AETC), November 1994.
12. Lloyd K. Mosemann, II, Deputy Assistant Secretary of the Air Force. "Action Memorandum of Air Force Policy on Programming Languages." Washington, D.C., August 1990.
13. Lyngbaek, Peter. "Object-Oriented Database Evolution." *OOPSLA Tutorial 10: Object-Oriented Database Systems*. October 1991.
14. Moyers, Anthony D. *An Object-Oriented Database Interface for Ada*. MS thesis, AFIT/GCE/ENG/93D-11, Air Force Institute of Technology (AETC), December 1993.
15. Object Design, Inc., Burlington, MA. *ObjectStore Administration and Development Tools* (2.0 Edition), October 1992.
16. Object Design, Inc., Burlington, MA. *ObjectStore User Guide* (2.0 Edition), October 1992.
17. Ricciuti, Mike. "Object Databases Find Their Niche," *Datamation*, 56–58 (September 1993).
18. Rosenberg, Dave. *ObjectStore and Ada*. Technical Report, Burlington, MA: Object Design, Inc., January 1992.
19. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

20. SunPro. *Sun Ada Programmer's Guide*. Sun Microsystems, Inc., Mountain View, CA, 1992.

21. Wessel, Chuck. "Itasca Functionality." Electronic Mail #9403072135.AA09095, March 1994.

Vita

Captain Stephen R. Lindsay was born in Belleville, Illinois on August 1, 1967. After graduating from Shawnee Mission East High School in Prarie Village, Kansas, he attended the University of Kansas, where in 1989 he received a Bachelor of Science in computer science and a regular Air Force commision. Capt Lindsay attended AFIT after a three-year tour as a systems analyst for the Space Surveillance Center in Cheyenne Mountain AFB, Colorado. His assignment upon graduation from the School of Engineering is the Satellite Control Simulation Division, Phillips Laboratory, Kirtland AFB, New Mexico.

Permanent address: 239 NW 41
Warrensburg, Missouri 64093

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Designing and Implementing an Ada Language Binding Specification for ODMG-93	5. FUNDING NUMBERS
6. AUTHOR(S) Stephen R. Lindsay	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/94D-16
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter WL/AAWA Wright Laboratory Wright-Patterson AFB, OH 45433	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Object-oriented database management systems (ODBMSs) represent the latest advancement in database technology, combining the reusability and maintainability of the object-oriented programming paradigm with the ability to efficiently store and retrieve a wide range of data types as well as code to manipulate stored data. Unfortunately, programmers developing software in the Ada programming language do not have the ability to interface to object databases without significant customized code development. One important reason for this has been the absence of a standard defining the constructs accessing ODBMS functionality.
This thesis documents the creation of an Ada language binding to the ODMG-93 standard for object database interfaces. Key portions of the binding are then implemented for the Itasca and ObjectStore ODBMSs written in Lisp and C, respectively. This work achieved the goals of a complete, portable, and transparent object database interface for Ada. To satisfy transparency a preprocessor was assumed to exist for both implementations, and its degree of involvement was directly proportional to the degree of strong typing of the ODBMS implementation language.

14. SUBJECT TERMS Object-Oriented Databases, Ada, ODMG-93, ODMG	15. NUMBER OF PAGES 132
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL